National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

The Enterprise Code Librarian

by

Enoch Chan ©

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Masters of Science

Department of Computing Science

Edmonton, Alberta
Fall 1992

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77278-6

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Enoch Chan

TITLE OF THESIS: The Enterprise Code Librarian

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material from whatever without the author's prior written permission.

(Signed) Enoch Chan

Permanent Address:
Box 60723, U of A Postal Outlet,
Edmonton, Alberta,
Canada. T6G 2S8

Date: July 20th, 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **The Enterprise Code Librarian** submitted by **Enoch Chan** in partial fulfillment of the requirements for the degree of **Masters of Science**.

JSchaeffer
Dr. Jonathan Schaeffer (Supervisor)

Bruce Cockburn
Dr. Bruce Cockburn (Electrical Engineering)

Duane Szafron
Dr. Duane Szafron

Date: July 3rd, 1992

Abstract

Enterprise is a programming environment for writing parallel/distributed software that is suitable to run on a network of workstations. The environment facilitates the rapid construction of distributed programs by taking care of issues such as low-level communication protocols, synchronization problems, and deadlock avoidance. An Enterprise program is written in familiar sequential language, which can be compiled for sequential execution usually without modifications. The parallelism of the program is specified separately by attaching templates, called assets, to sequential modules through an Enterprise graph. Each asset represents a high-level parallelization technique that is frequently used by many distributed programs. The term *asset* comes from an analogy between a distributed program and the working of an organization. The low-level communication code is generated by the system automatically according to the specified asset types. The separation of the specification of a program's source code and its parallel structure allows the program to be restructured easily without changes to the source code. This also allows a program to be adapted easily to the changing resources available in a workstation environment.

This thesis presents the work on the design of the system architecture of Enterprise, the design and implementation of the Enterprise *code librarian,* and the Enterprise *pre-compiler.* The design of the architecture allows different components of the system to be implemented individually. Currently, several components have been implemented to allow Enterprise applications to be developed and tested. The Enterprise *code librarian* is designed for managing the source and object code of Enterprise applications. Since Enterprise applications are designed to run on a heterogeneous network of workstations, the *code librarian* takes this into account and provides a makefile generation utility to maintain multiple executable files for a variety of architectures. The Enterprise *pre-compiler* is used for converting sequential calls into remote procedure calls, changing return statements into reply statements, and substituting the function declarations with a suitable format to allow remote invocations. Several applications have been developed using the Enterprise environment. Experimental results show that Enterprise offers a cost effective and easy to learn method for the rapid construction of distributed software.

Contents

1 **Introduction** 1
- 1.1 Introduction 1
- 1.2 The Problem 2
- 1.3 The Enterprise Approach 3
- 1.4 Contribution of the Thesis 7
- 1.5 Organization of the Thesis 8
- 1.6 Conclusion 8

2 **Reviews of Parallel Programming Tools** 9
- 2.1 Introduction 9
- 2.2 Types of Parallel Programming Tools 11
  - 2.2.1 Expressing Parallelism in Programming Languages 11
  - 2.2.2 Dependency Analysis Tools 13
  - 2.2.3 Automatic Parallelizer 13
  - 2.2.4 Integrated Parallel Programming Environment 14
  - 2.2.5 Other Tools for Workstation Environment 15
- 2.3 Support for Conceptual Models 16
  - 2.3.1 Task-Oriented Models 16
  - 2.3.2 Task-Oriented Programming Tools 17
  - 2.3.3 Data-Oriented Models 18
  - 2.3.4 Data-Oriented Programming Tools 20
  - 2.3.5 Object-Based Models 22
  - 2.3.6 Object-Based Programming Tools 23
- 2.4 Other Issues in Parallel Programming 25
  - 2.4.1 Explicit and Implicit Parallelism 25
  - 2.4.2 Expressiveness 26
  - 2.4.3 Ease of Programming vs. Full Programming Control 26
- 2.5 Conclusion 27

3 **The FrameWorks System** 28
- 3.1 The FrameWorks Model 29
  - 3.1.1 Modules 29

3.1.2 Templates 31
3.2 Developing Applications using FrameWorks 32
3.2.1 Mod_Craft: The Graphical User-Interface 32
3.2.2 Compiling FrameWorks Applications 33
3.2.3 Code Generation in FrameWorks 34
3.3 Debugging facility 35
3.4 Communications Between Different Components 35
3.5 Limitations of FrameWorks 36
3.6 Conclusion 38

4 The Enterprise System **39**
4.1 Introduction 39
4.2 Module Calls 40
4.3 Module Roles and Assets 42
4.3.1 Individual 43
4.3.2 Line 43
4.3.3 Pool 44
4.3.4 Contract 44
4.3.5 Department 45
4.3.6 Division 46
4.3.7 Service 46
4.3.8 Other Asset Kinds 47
4.4 Enterprise Graphs and Parallelism 48
4.5 Enterprise Program Construction 49
4.6 The Architecture of Enterprise 51
4.6.1 Interface Manager 54
4.6.2 Application Manager 54
4.6.3 Code Librarian 55
4.6.4 Execution Manager 56
4.6.5 Resource Secretary 56
4.6.6 Monitor/Debugger Manager 57
4.7 System Implementation 57
4.8 Conclusion 58

5 Source Code and Object Code Librarian **60**
5.1 Introduction 60
5.2 Management of Source and Object Code 61
5.2.1 Using Architecture Keywords 62
5.2.2 Using Architecture-Specific Directories 63
5.3 Makefile Management for Enterprise Applications 64
5.4 Creating the Sub-directories and Imakefiles 66
5.4.1 Generation of the Imakefiles 67
5.4.2 Contents of the Imakefile 69

5.5 Limitations of the Gen_make Utility 71
5.6 Conclusion 72

6 Implementation of the Enterprise Compiler **74**
6.1 Introduction 74
6.2 Comparison between FrameWorks and Enterprise 75
6.2.1 The FrameWorks Approach 76
6.2.2 The Enterprise Approach 77
6.3 Interface to the Compiler 78
6.3.1 First Pass of the Compiler 78
6.3.2 Second Pass of the Compiler 83
6.4 Implementation Details 84
6.4.1 The GNU C Compiler 84
6.4.2 Implementation Procedures 84
6.4.3 Recognition of Function Names and Function Calls 85
6.4.4 Recognition of *RETURN* Statements 86
6.4.5 Recognition of Parameter Types 87
6.4.6 Another Approach to Handling Parameter Types 89
6.4.7 Recognition of Variables for Delay Blocking 90
6.4.8 Interface to ISIS Code 90
6.5 Semantic Issues 96
6.6 Conclusion 98

7 Conclusions and Future Research **100**
7.1 Introduction 100
7.2 Thesis Summary 101
7.3 Recommendations for Future Research 103

Bibliography **106**

A The Configuration File of Gen_make **110**

B The Generated Imakefile **111**

C The Generated Check File **113**

D Example of User Source Code **114**

E Example of Enterprise Inserted Code **116**

Chapter 1

Introduction

1.1 Introduction

As local area networks of personal workstations become increasingly popular these days, their main usage is the sharing of information and peripherals, and the facilitating of communication among users. In this kind of computing environment, computation-intensive work is usually done on a personal workstation. While more powerful personal workstations are appearing on the market in a rapid pace, there is still a large number of applications which require more processing power than a single workstation can offer. These applications often require the power of large time-sharing mainframe or parallel computers to run them effectively. However, this power is also available on a network of workstations. In fact, a network of workstations possesses a large amount of combined processing power which may even exceed that of some supercomputers. In spite of the availability of this power, it is not easily utilized by the users on the network.

A major obstacle in utilizing this processing power lies in the complexity of writing parallel/distributed applications. To write an error-free parallel application, a programmer has to take care of issues such as synchronization and communication between processes. Much recent research (for example ParaScope [6], PIE [41], and Paralex [4]) is targeted toward a better and integrated programming environment for developing parallel/distributed applications, such that this processing power can be harnessed in an easier fashion. Among these programming environments, the FrameWorks system is one of the earlier systems having the above objectives in mind [44, 43, 45]. The development of a new parallel and distributed programming tool which allows users to utilize the shared processing power available on a network is the concern of this thesis.

1.2 The Problem

It has long been realized that parallel/distributed software offers many advantages which cannot be found in comparable sequential software. For example, a distributed program is likely to run faster because of the opportunity to utilize the additional shared processing power. The use of distributed processing may also eliminate the need for expensive, but high-performance, specialized computers, the only source for the required processing power in the past. Replacing these specialized computers with workstations allows the same computing environment to be used regardless of problem size, and the cost of hardware maintenance to be reduced.

On the other hand, the use of distributed software is not without drawbacks. The development of distributed software is much more complicated than sequential software. The design, implementation and testing of a distributed application usually requires a large amount of time and effort from the programmers. Although many parallel/distributed algorithms have been developed, few of them are actually being used in practice. Actually, the design of an algorithm may represent only a small portion of the overall cost of development. The rest of the costs are often due to issues such as deadlock avoidance, synchronization, communications, heterogeneous computers and operating systems, and the difficulties involved in testing and debugging the software due to non-deterministic executions.

The utilization of the shared processing power in a network also poses some additional problems. A distributed programming environment has several unique problems which do not appear in tightly coupled multi-processor computers, such as multiple-instruction multiple-data-stream (MIMD) computers:

1. The run-time environment of distributed software changes frequently. The processors and their capabilities available to an application may vary from one execution to another. In addition, the cost of sending a message from one workstation to another depends largely on the amount of traffic on the network.
2. The execution of a distributed application has to consider the ownership of individual workstations on the network. Usually, the owner of a workstation would allow other users to use his computer when it is idle, but there may be situations in which he would prefer to be the only user of the machine.
3. The use of a computer network implies a high inter-process communication cost. This higher communication cost restricts the types of parallelism which may be efficiently implemented. For example, most small-grain parallel programs cannot take advantage of the shared processing power on a network because of the high communication overhead.
4. The low-level communication codes are usually hard to master for non-expert users. Most users do not want to become experts in networking or low-level communication packages only for exploring the potential parallelism. Usually, they would prefer to concentrate on the functionality and correctness of their application, rather than focusing on the low-level communication protocols.

1.3 The Enterprise Approach

This thesis is concerned with the design and implementation of a distributed programming environment, called *Enterprise,* that is suitable for shared processing in a workstation environment. Enterprise represents the complete redesign and re-engineering of the FrameWorks system. While Enterprise uses many of the ideas developed in FrameWorks, there are major changes in the user's view of a program and refinements in the tools (such as the user interface) available to the user.

Enterprise provides a programming environment for designing, coding, debugging, testing, profiling, and executing programs in a distributed environment. Enterprise applications run on a network of workstations trying to utilize the combined processing power and resources. Each Enterprise application is divided into two parts: the program source code modules, and a computation graph. The first part is the program source code which looks like familiar sequential code. The second part is the graphically specified parallel structure of the application. The system automatically generates the communication codes used by the specified structure and inserts them into user's source code. This arrangement ensures the correctness of the communication code, and allows the rapid construction of distributed programs.

Enterprise also keeps in view the constraints described in Section 1.2 such that they will not be violated. The Enterprise system is designed with four basic principles in mind [12, 49]:

1. to provide a simple high-level mechanism for specifying parallelism that is independent of low-level synchronization and communication protocols,
2. to provide transparent access to heterogeneous computers, compilers, languages, networks, and operating systems,
3. to support the parallelization of existing programs to take advantage of the investment in the existing software legacy, and
4. to be a complete programming environment, eliminating the overhead that arises from switching between it and other programming environments.

There are several distinct features that separate Enterprise from other parallel and distributed programming tools:

1. Enterprise programs are written in a familiar sequential language that is augmented by new semantics for function and procedure calls. The new semantics allow a function call made to a remote function, which may be running in parallel on another machine,

to appear as a normal sequential call. The distributed code used for handling the communication or synchronization between processes will be inserted by Enterprise.

2. Enterprise supports a set of commonly used parallel structures and generates the low-level communication code accordingly. These structures correspond to the regular parallelization techniques, such as pipelines and master/slave relationships, which could be used to structure most of the large-grain parallel programs. The desired technique is specified graphically through a graphical user interface, and is independent of the user written code. The de-coupling between the user's code and the desired parallelization technique allows an application to be restructured without changes to the original source. This also has the added flexibility that applications can be adapted easily to the changing resources available on a network.

3. Enterprise uses an analogy between the structure of a parallel program and the structure of an organization to simplify the way parallelism is expressed. This analogy is used because of the inherently parallel and hierarchical structure of an organization, and a consistent set of names can be used to describe the parallel constructs. For example, traditional terminologies such as:

   pipelines, masters, slaves, etc.

   can be replaced by:

   divisions, departments, individuals, etc.

   This analogy is also known as anthropomorphic programming which has both advocates [8] and detractors [17] in computing science. In fact, the use of analogies has proven useful in object-oriented programming.

4. Enterprise supports the dynamic distribution of work in environments with changing resources. The dynamic distribution of work is supported through some of the parallelization techniques defined in Enterprise. For example, a *contract* may be used to

distribute work to a changing number of identical co-workers (processes). A contract could use as many idle workstations as available to complete the assigned work. For instance, when the network is heavily loaded, only a handful of workstations may be available to fulfill the contract. As the load decreases, many more workstations could be used by the contract to finish the task.

5. Enterprise uses a hierarchically structured computation graph to specify the parallelization techniques used by an application. This graph is constructed in a different way than the graphs used in other parallel programming tools. The user of Enterprise does not construct the graph by connecting nodes (processes) by arcs (communication paths). Instead, an Enterprise graph always starts with a single node in the diagram. The user edits the diagram by either coercing or expanding the node. Coercing a node allows the communication structure between neighbors to be expressed. Expanding a node allows the hierarchical structure of the application to be explored.

6. Enterprise allows users to have complete control on the mapping of processes to processors. As pointed out by Jones and Schwartz [26], hiding the underlying hardware details from the users may result in major performance degradation of distributed systems. Enterprise users can control the mapping either by doing a partial mapping, doing a complete mapping, or letting the system to handle the mapping on its own. Similar to the use of parallelization techniques, the processes to processors mapping is independent of the user-written source code.

In short, Enterprise provides a high-level programming tools for developing parallel and distributed programs. Enterprise users do not deal with the low-level communication or synchronization code when writing distributed programs. Instead, Enterprise inserts the code needed for handling distributed processing into user's source code. Enterprise starts the execution of a program by dynamically assigning processes to processors, and establishing the necessary communication channels. Processes will be running in the background on remote workstations to make use of the additional processing power. Furthermore, Enterprise also monitors the global status of the network to perform load balancing. Enterprise processes running on one heavily load workstation could be moved to a less busy one without affecting the execution of the program. In this way, the normal users of the workstation will not be disturbed by Enterprise processes.

1.4 Contribution of the Thesis

Enterprise is an ongoing research project at the University of Alberta on the development of an easy to use programming environment for developing distributed programs. This thesis represents part of the research work being done on the development of the system. This thesis will be focused on the implementation of the *code librarian* and the *compiler* used by Enterprise. In this section, we outline the major research contributions of the thesis:

1. Describes the Enterprise programming model used for distributed applications. This model separates the specification of the program source code from that of the program's parallel structure, encouraging users to experiment with different forms of parallelism.

2. Designs and specifies the overall architecture of the system such that the functional requirements of each subcomponent can be identified and implemented individually.

3. Designs and implements the Enterprise Code Librarian. The librarian is designed for managing the source and object code of Enterprise applications. It uses *Imake* [19] to generate makefile templates, and takes into account the possible heterogeneity of computers available on a network of workstations.

4. Develops the compiler used by Enterprise to insert codes for handling distributed operations into user programs. This compiler is responsible for translating function calls to a modified/extended form of remote procedure calls. It is also responsible for packaging the parameters of a call into messages being passed from one workstation to another.

1.5 Organization of the Thesis

The main focus of this thesis is on the design and implementation of the source and object code librarian and the compiler used by Enterprise. The rest of the thesis is organized as follows:

Chapter 2 of this thesis discusses the issues involved in parallel programming, and reviews some existing parallel programming tools.

Chapter 3 gives an overall description and a review of the FrameWorks System.

Chapter 4 provides an overview of the Enterprise programming model, and presents the overall architecture of the system.

Chapter 5 gives a discussion of issues involved in managing the source and object code of Enterprise applications. This chapter also outlines the implementation of the Enterprise Code Librarian.

Chapter 6 describes the implementation of the Enterprise compiler used for inserting distributed code into user programs. Also, semantic issues resulted from overloading the syntax of function and procedure calls are discussed.

Chapter 7 provides a summary of the thesis, and future research topics will be discussed.

1.6 Conclusion

Together with the growing popularity of workstation environments, there comes with an increase in the availability of idle processing power. Although this power could be used by distributed programs, it is not easily utilized by general users. The development of distributed software is often complicated by low-level communication details. In general, users would prefer to avoid the extra complexity involved in developing distributed programs. As a result, a large portion of this processing power is being wasted. This thesis describes a new programming environment which can be used to bridge the complexity gap between distributed and sequential programming so that the unused power can be more profitably utilized.

Chapter 2

Reviews of Parallel Programming Tools

2.1 Introduction

In recent years, many tools have been developed to ease the writing of new parallel programs as well as to parallelize sequential programs. These tools range from pure dependency analysis tools to integrated parallel programming environments. In effect, these tools are used by a programmer to aid in transforming a problem solution from one's "mental model" into a "second model in the form of workable code" [37]. This process of transformation was once very difficult because of the increased complexity involved in writing parallel programs. The programmer often needs to deal with low-level operations to handle issues such as communication, synchronization, and deadlock avoidance among parallel processes. The processes could be running either on different processors in a parallel computer or on workstations in a distributed environment. Furthermore, the decision on optimal process partitioning, process to processor mapping, and process identification are also important factors which affect the efficiency of a parallel program [24].

With the introduction of parallel programming tools, the user can write parallel software at a much higher level than before. These tools are designed to handle most of the low-level operations for the user. These tools also provide a more stable programming environment for writing parallel software. It is generally agreed that architectures of parallel computers change more rapidly than software programming environments. If an application is developed in the context of the underlying physical computation model, it is most probable that the application will have to be re-engineered every time it is ported to another parallel architecture. As Ahuja [1] pointed out, sometimes it may be necessary for the implemented application to be "conceptually reformulated" for it to run on other parallel computers. Writing applications by using parallel programming tools increases the portability of the application programs. Moreover, programmers often prefer a "familiar and convenient" software development environment with reasonable performance over an environment which gives maximum performance but require a significant amount of additional programming effort [38].

From another perspective, these tools provide the user an abstract view of the underlying parallel architecture. Many of these tools encourage the user to write applications in terms of an abstract view, which can be mapped to different underlying physical hardware, to reduce the hassle of porting the applications. This abstraction, which may be called a *conceptual model* or a *virtual machine*, shifts the focus of parallel programming from dealing with hardware details to the design of the overall parallel structure of a program. This is similar to sequential programming where the low-level operations, such as reading a byte form the disk, are handled by the operating system. This shifting of focus not only relieves the user from programming hardware details, but also allows the program to be expressed in its most natural way [51].

In the following sections, the types of tools available today, the conceptual models that they support, and their strength and limitations will be discussed.

2.2 Types of Parallel Programming Tools

Many different kinds of parallel programming tools, such as new parallel programming languages or integrated parallel programming environments, are available today. Since the design and implementation of these tools are closely related to the type of parallel programming languages that they support, a discussion of how parallelism is expressed in programming languages will be presented first.

2.2.1 Expressing Parallelism in Programming Languages

Parallelism can be expressed in a programming language either by using new parallel programming languages, or by adding high-level parallel constructs as an extension to existing sequential languages [13, 36]. The latter approach may be implemented by creating communication libraries such as NMP [32] or ISIS [7]. Yet another way of supporting parallelism is by using an optimizing compiler or an automatic parallelizer to perform the parallelization of sequential programs. Each of these approaches has its own merits and problems.

On the one hand, new parallel programming languages, such as Linda [1] and Strand 88 [22, 21], have the advantage of employing the latest concepts and new ways of specifying parallel constructs in the languages. On the other hand, new languages are often incompatible with old sequential languages. This makes the parallelization or transformation of existing sequential programs an error-prone and time-consuming task. In addition, new programming languages often demand a large amount of learning time from the programmer. For example, the strength of Strand 88 is its recursive nature, which allows it to express the creation of dynamic processes in a natural way. However, Strand 88 is also hard to learn and to program with because it requires the programmer to think recursively [13].

The language extension approach is more compatible with traditional sequential programming languages, and is usually easier for the programmer to master. In addition, because parallel programs are written in a language familiar to the programmer, it opens up the opportunities to fine tune the performance of the programs. By using the language extension approach, not only are new parallel programs easier to write, but it also increases the reliability of the programs. The simplicity of a language is an important feature according to Hoare, who has quoted that "if programs are to be reliable, the languages they are written in must be simple to understand and use" [2, 18].

Although the language extension approach is easier for users to understand, it may be hard or even impossible to express some of the commonly used parallel constructs. One commonly used parallel construct, often missing in the tools using such an approach, is the tree-like structure. This structure is being used widely in parallel search, divide-and-conquer, and other classes of algorithms.

According to Chang & Smith [13], the language extension approach can be further classified into:

1. pre-processors, e.g. FORCE [28],
2. pre-compilers/automatic parallelizers, e.g. the Cyber 205 Fortran compiler [16], or
3. language features, e.g. ADA.

The easiest approach to implement is the pre-processor approach. It uses compiler directives or macro-processors to generate the necessary parallel code for the application, and then uses a sequential compiler together with some low-level parallel library routines (such as NMP or ISIS) to produce the target machine code. The automatic parallelizer is more expensive to build as it requires dependency checking and knowledge of code segments (i.e. the semantics) in the context of the original program. Finally, the language features approach is even more expensive to implement for it usually requires new compilers rather than using existing compilers as one of its building blocks.

Having discussed how parallelism is supported in programming languages, the following sections describe the different types of available tools. There are generally three types of parallel programming tools: (1) pure dependency analysis tools, (2) automatic parallelizers, and (3) integrated parallel programming environments.

2.2.2 Dependency Analysis Tools

Pure dependency analysis tools are generally targeted toward making the parallelization of an existing sequential program easier. The parallelization of a sequential program generally involves repartitioning and remapping. Although it may not be difficult to repartition a sequential program written in a high-level structural language, problems may appear if the program is written in older languages such as Fortran. Several tools are designed to uncover the design of a sequential program and display it graphically for further analysis. Examples of these tools are E/SP, MIMDizer, and PRETS [24].

The programmer typically uses the dependency analysis tools in an iterative manner. An existing sequential program is first passed into a dependency analysis tool to collect initial dependency information. This information is gathered by performing static data and control flow analysis on the program. Some of the dependency analysis tools also collect run-time statistics to aid the analysis process. The collected information will be used to identify parallelization possibilities and display them to the programmer through a graphical display.

The dependency analysis tools usually use a conservative approach in identifying the parallelization possibilities because it is generally impossible to determine at compile time if a data dependency exists between two variable references [3]. The programmer may supply additional information to the tool to "replace conservative assumptions" about the program [24]. The program can then be restructured and re-analyzed according to the collected information. This restructuring and redesigning process may be repeated until a suitable parallel structure for the program is found. Because the dependency analysis tools are not automatic parallelizers, the programmer must perform the final parallelization step manually to produce a parallel program.

2.2.3 Automatic Parallelizer

Another type of parallel programming tool is the automatic parallelizer or the optimizing compiler. An automatic parallelizer performs all the necessary dependency analysis and optimization of a sequential program for the user. Although this approach seems to be attractive for converting existing programs, it may be unable to obtain the maximum performance from a given parallel processing environment. Another limitation of an automatic parallelizer is that the programmer is given few opportunities for exploiting different forms of parallelism and in fine tuning their parallel programs [3]. Examples of automatic parallelizers are the EXPRESS C automatic parallelizer [13], the Alliant FX Fortran compiler [24], and the Cyber 205 Fortran compiler [16].

An optimizing compiler is relatively easy to use, but the programmer often has to be concerned about the underlying hardware architecture to use the compiler effectively. For example, the Cyber 205 Fortran compiler compiles an otherwise sequential program and outputs the parallelized target code for the Cyber 205 supercomputer. The compiler tries to parallelize the sequential program by doing dependency analysis and locating loops that are parallelizable. The quality of the target code depends largely on the structure of the sequential program. Often a large performance improvement can be realized simply by doing subscript alignment in a do-loop [3], by distributing loops (i.e. dividing a non-parallelizable loop into multiple loops where some of the loops may be parallelizable), by making a scalar variable into an array (breaking scalar dependencies), or by exchanging the inner-loop with the outer-loop of a nested do-loop (changing the granularity of the program). The performance improvement is due to new parallelization opportunities resulting from restructuring the program, and maximal use of hardware features such as using the hardware pipeline and reducing the amount of memory paging activities. The use of an optimizing compiler tends to mask the underlying hardware from the user. However, if the user does not understand the hardware, he will not know enough to restructure his program to get better performance.

2.2.4 Integrated Parallel Programming Environment

The third type of parallel programming tools is the integrated parallel programming environment. These environments provide the programmer a set of tools to edit, test, debug and visualize the execution of a parallel program. While the set of tools in an integrated environment often share a consistent and uniform set of user interfaces, some environments provide them in the form of a uniform graphical display. A structural graph, in the form of a control flow graph or a data dependency graph, is used to define the parallel structure of a program. The run-time behavior of the program can be visualized by animating the defined structural graph. The visualization of program execution is particularly important, because it provides an easy alternative for the programmer to identify performance bottlenecks and potential synchronization problems of parallel programs.

In addition to the advantage of visualizing the execution of parallel programs, most of these tools, such as Paralex [4] and FrameWorks [44, 43, 45], allow the independent definition of functional modules and the parallel program structure. Every functional module has a well defined interface (i.e. the function's header) to other modules; also, the module does not assume any synchronization properties in its source code. This feature has the added flexibility of being able to restructure a finished parallel program and to experiment with different forms of parallelism to achieve a better performance. The separation of the two definitions also enhances the reusability of the functional modules [10], and helps to make a parallel program become more portable. The programming models represented by these tools are usually conceptual models rather than a specific physical model. Most of the newly developed tools fall into this particular type.

2.2.5 Other Tools for Workstation Environment

In the above discussion, we did not distinguish between the tools which are designed for shared memory parallel computers from the tools designed for distributed memory parallel computers or even a network of workstations. The tools discussed so far are used for writing efficient parallel programs, and every tool tries to fully utilize the available processing cycles in the parallel computing environment. There exists, however, yet another kind of tool which tries to fully utilize the computational power available in a workstation environment. These tools, such as the Engineering Network Computer (ECN) [23], the Butler system [35] and WORM [42], usually provide to a user a mechanism for executing processes transparently on a remote idling workstation [23, 35].

The difference between using these tools and simple remote procedure calls is in the selection of a workstation. In using remote procedure calls, for example *rcmd* in UNIX, the user has to explicitly specify which workstation is to be used. However, a tool such as the ECN will automatically choose a workstation for the remote process based on some pre-defined criterion. Some of these tools, such as WORM, even provide a mechanism to migrate a remote process from one heavily loaded workstation to another workstation [42]. These tools are useful in reducing the overall response time of a workstation-based computing environment, but they are not designed for developing large parallel programs which often require a huge amount of computation power.

2.3 Support for Conceptual Models

We have discussed so far the types of parallel programming tools and the merit of programming in terms of conceptual models. In this section, we will discuss the different conceptual models available and the specific support available through parallel programming tools.

In general, there are three different conceptual models available for writing parallel programs: task-oriented models, data-oriented models, and object-based models [37]. Although there are several conceptual models available, most parallel programming tools have direct support for only one model. Sometimes additional features are added to the tools such that some degree of freedom can be achieved.

2.3.1 Task-Oriented Models

In the task-oriented model, a parallel program is viewed as a collection of serial processes and each of these serial processes has its own unique thread of control. When some of the serial processes are replicated and executed in parallel, parallelism can be achieved. Some basic forms of this model, the ***independent processes model*** and the ***cooperating processes model,*** are supported by programming languages such as Algol 68, Smalltalk, Occam, Ada, and PL/1 [37].

In the independent processes model, each process executes independently and terminates without any interaction between concurrent processes. This is similar to the *fork* and *join* used in many UNIX systems. Typically, a parent process forks off a number of child processes. Each of these child processes may continue to execute without any interaction with other child processes. Synchronization points, such as a *join*, may be used to control the flow of the program. When the child processes are terminated, a join may be performed and the control is returned back to the parent process. In the cooperating processes model, concurrent processes can communicate with other processes by means of message passing or shared storage area. A parallel program in the task-oriented model can be represented by a directed graph with each node representing a task or a procedure call, and each arc representing an operation which advances the state of the program. Since the independent process model is a subset of the cooperating process model, the latter is usually the model being implemented [37].

2.3.2 Task-Oriented Programming Tools

PISCES 2 (Parallel Implementation of Scientific Computing Environments) [38] is one of the tools which uses the task-oriented model. The main concept of PISCES 2 is to allow the programmer to write programs in terms of a well-defined virtual machine. The virtual machine is independent of the underlying hardware such that programs should be easy to be ported to other computers. A parallel program in PISCES 2 is viewed as a collection of tasks, and the tasks are grouped into different clusters. The clusters in a program run in parallel and communicate through asynchronous message passing. The tasks in a cluster also run in parallel, but they communicate through shared variables. An individual task can be further divided into "forces" which represent parallel running "code segments" such as parallel loops or subroutines. PISCES 2 is implemented on the Flexible FLEX/32 system, a MIMD computer, as a set of Fortran extensions.

SCHEDULE [24, 38] and POKER [13, 38] are systems similar to PISCES 2. POKER was originally designed for use on the CHiP computer, but it was later used as an environment for programming different MIMD computers. While POKER has a graphical interface for user interactions, PISCES 2 requires users to structure their applications through textual interfaces. SCHEDULE is a set of Fortran extensions which allow a program to be partitioned into independent tasks. The programmer specifies the dependencies of the tasks and SCHEDULE uses a global queue to control the execution sequence of the tasks. The process (task) to processor mapping is handled automatically by SCHEDULE. If possible, the tasks are scheduled to run in parallel and communicate through shared variables. SCHEDULE also supports a visualization tool for monitoring the execution of a program.

PAT (Parallelizing Assistant Tool) [3, 24] is a collection of tools which aid Fortran programmers in writing, debugging, and fine tuning applications. An interactive parallelizer is used in PAT to perform control and data flow analysis. The parallelizer uses the information to make suggestions concerning potential modifications for enhancing the parallelism in the program. PAT also uses a static analyzer to simulate the execution of the program to locate potential errors such as deadlock conditions. A graphical user interface is used to show the information gathered form the parallelizer and the static analyzer. The user interface also allows the programmer to restructure his program interactively.

E/SP [24] is a semi-automatic Fortran parallelizer. A Fortran program is displayed as a graphical hierarchical dependency graph. E/SP uses a graphical user interface to allow users to interactively restructure his program. Code segments that are not possible for E/SP to perform automatic parallelization will be displayed as highlighted icons on the dependency graph. The programmer can decide on actions such as separating loops, or reorganizing nested loops to increase parallelism. E/SP then performs a "source-to-source" translation to create the parallel program source for the Fortran VII compiler, an optimizing compiler for the Concurrent 3200MPS computer.

2.3.3 Data-Oriented Models

A parallel program written in the data-oriented model is parallelized by concurrent execution of an operation on multiple data items [37]. This is similar to the computational model of a SIMD computer. Different from the task-oriented model, where a parallel program has multiple threads of control, a data-oriented parallel program exhibits only one thread of control. When the size of the data items is large enough for parallelization, the program diverges temporarily into parallel actions to operate on the data subsets, and converges when the actions finish [37].

A data-oriented program can be expressed as a data dependency graph with each node representing a computation unit and each arc representing the flow of data. Furthermore, a data-oriented program can belong to either one of the *demand-driven, data-driven,* or the ***protocol-specified models*** [25, 5, 10, 4]. In the demand-driven model, the execution of a program is initiated by having the terminating node to send a *request for data* to the initial node and the initial node responds [25]. The terminating node and the initial node in the dependency graph are represented by the sink node and the source node respectively. A source node is a node which has no incoming edges, and the sink node is a node which has no outgoing edges.

In fact the sink node, in the demand-driven model, does not send its demand for data request directly to the source node. The sink node first sends the data request to its parent in the dependency graph. The request is then propagated up toward the source node, and finally the result of computation is propagated back down to the sink or the terminating node [25].

The data-driven model [5] takes an opposite approach to the demand-driven model. The computation of the data-driven model is initiated when the starting node sends data or results to its children, and eventually the computation is ended when a result is received at the terminating node. A computation unit (i.e. a node) is never called or activated directly by another computation unit. The computation unit is activated by sending data items to it. The advantage of using the data-driven model over the demand-driven model is the reduction in the total number of messages. Since the data request messages are no longer necessary, for a similar data-dependency graph, the data-driven model uses only one half of the messages used by the demand-driven model [25, 51].

Whereas the data-driven model has the advantage of using fewer messages, there may be situations where the demand-driven model provides a more natural way of expressing the structure of a program. The latter is more general for it can support different forms of parallel evaluations. Both "call-by-value" and "call-by-need" function evaluations are supported in the demand-driven model, but only "call-by-value" evaluation can be naturally supported by the data-driven model [25]. "Call-by-value" function evaluation means the arguments of a function are evaluated before the function is executed; "call-by-need" function evaluation means that the arguments are not evaluated until they are needed in the function.

The protocol-driven model, used by CODE [10, 11] and Paralex [4], combines the advantages of the two data-oriented models. The protocol-driven model allows the programmer to specify both data-driven and demand-driven dependencies in a data dependency graph. The programmer specifies a "firing rule" for every edge in the graph. The rule is used to describe the type of dependency associated with an edge. In CODE, an edge can represent either a data, a demand, an exclusion, or a control dependency [10].

2.3.4 Data-Oriented Programming Tools

CODE (Computation-Oriented Display Environment) [10, 11] is designed to provide an "unified approach to parallel programming." It tries to combine the results of software engineering and visual-programming research to help write parallel programs. CODE is a data-flow oriented programming environment, in which a parallel program is divided into two different basic elements. The first element is a set of computation units, each containing a functionality and a firing rule. The functionality of a computation unit can be viewed as the computation carried on the input data, and the firing rule specifies the condition of when the computation unit should be executed. The second element consists of the dependency relations, which specify the dependencies among different computation units. The dependency relations can be any one of data, demand, mutual-exclusion, or control dependency. A dependency graph is used to define the dependency specifications of a program. A node in a dependency graph can be either a computation unit or a sub-graph. An edge in the dependency graph shows the dependency between various nodes. Finally, CODE encourages the programmer to design his programs in a "top-down" manner by writing them as "hierarchical graphs".

In a CODE program, the dependency specification is separated from the computation unit specification, and the firing-rule specification is separated from the functionality specification in computation units. Because of these separations, reusability of the source code and design modules is strongly enhanced [10]. CODE is frequently being used with ROPE (Reusability-Oriented Parallel Programming Environment) [11] which is designed to select and reuse CODE modules. Currently, CODE supports four programming languages: Ada, C, Fortran, and Pascal. One special feature of CODE is that the programmer writes only the sequential algorithm code and CODE generates the function's header. However, CODE does not provide mechanisms for dynamically creating processes at run-time.

DGL (Directed-Graph Language) [25] is another tool which uses the data-oriented model. An application in DGL is composed of a small number of large-grain function modules written in a traditional sequential language. Large-grain functions are used because of the high overhead involved in transmitting messages across a network. Function modules of an application are connected by a directed-graph language to specify the dependencies among the modules. A dependency graph can be used to represent the overall structure of a program. A node in the graph corresponds to a function module and an edge corresponds to the dependency between the modules. The distributed execution of a program is controlled by using the demand-driven approach. The DGL environment uses implicit parallelism such that the distribution of processes is transparent to the programmer. A session manager in DGL provides the programmer an interface for programming, debugging, visualizing, and executing programs.

LGDF (Large-Grain Data Flow) [5] is an environment for writing parallel Fortran programs using macro expansion and the data-oriented approach. The LGDF environment combines the data flow concept and sequential subroutines to produce a parallel program. A subroutine in LGDF is typically 5 to 50 lines of Fortran code. LGDF is data-driven as the subroutines are not called directly for execution, but are activated by the arrival of data items. In other words, if the value of all the arguments of a subroutine are available, the subroutine starts to execute. A LGDF program can also be expressed as a hierarchical data flow graph. The nodes in the graph can either be functional modules or other sub-graphs. The edges in the graph represent dependencies and data paths between the nodes, and shared memory is used in LGDF to implement the data paths. One serious drawback of the LGDF environment is in the construction of the dependency graphs. In LGDF, for each argument passed between two nodes, a new edge has to be constructed to specify the data dependency. Since a subroutine usually takes more than one input/output argument to execute, multiple edges are often required in a LGDF graph to connect two individual nodes. This may result in a complicated or an inconsistent dependency graph for even a relatively simple program.

Paralex [4] is another programming environment that uses the data-driven approach. Different from other data-oriented models, a node in a Paralex dependency graph can either be a computation node or a filter node. The computation node starts its execution when the required arguments have arrived. The filter node is used to allow data values to be examined before they are actually transmitted to reduce the amount of communication overhead. Paralex is used for developing parallel C programs by inserting ISIS library calls [7] to handle remote procedure calls. A graphical user interface is provided in Paralex to ease the development process of a program.

2.3.5 Object-Based Models

Object-based models are probably the newest conceptual model available to parallel programmers. The idea behind this model is similar to that of object-oriented programming in writing sequential programs. An object-based program is composed of modularly decomposed units. These units are independent and self-contained entities which may contain data, operations or both [36]. The basic function of such an entity is to interact with other entities by sending and receiving messages.

Although this model was developed along the lines of object-oriented programming, not all tools which fall into this categories are truly object-oriented. An object-oriented model requires an object to have an external (public) and an internal (private) views. These views are similar to the use of abstract data types where data can only be accessed through a well-defined interface. Furthermore, the model also uses the notion of *member functions,* which allows the same interface (external view) to be shared by different object types (classes of objects) to perform different actions. Tools such as FrameWorks [44, 43, 45] and Enterprise [12, 49] can be classified as object-based models for they allow the programmer to define a functional module which may contain its own data set. However, they do not have the support for designing *high-level object types.* Similarly, Linda provides a means for doing abstraction and system decomposition [37], but it is not an object-oriented parallel programming language. It lacks the ability to allow an interface to be shared among different abstractions. Each abstraction requires its own unique interface to be used for accessing the defined data or functions.

2.3.6 Object-Based Programming Tools

FOG (Fragmented Object Generator) [31] is built on the object-oriented model, and is mainly used to write distributed systems and distributed applications. A program in FOG is composed of different objects, called *fragmented objects.* These objects have both an external view and an internal view. The external view is the interface which may be used by other objects to communicate with the *fragmented object.* The internal view is the implementation of the object itself. A *fragmented object* may be shared or used by several objects at the same time. The *fragmented objects* may be created dynamically during run-time by a "binding" procedure similar to a "constructor" in the language C++.

FOG is implemented as an extension to the C++ programming language. A set of low-level communication objects are also implemented as a tool set to handle communications between distributed processes. The communication objects of similar functions are "organized as a class hierarchy, with the same interface" [31]. The object-oriented approach used by FOG is a powerful approach because additional objects, such as shared memory for distributed applications, may be implemented by using low-level objects if they are needed. As FOG is relatively new in the group of parallel programming tools, not many types of low-level communication objects are currently available. The lack of these objects might cause difficulties in developing new programs and in converting existing programs.

Linda [1] is a new parallel programming language to the extent that the "pre-processor or compiler recognizes the Linda operations, checks and rewrites them on the basis of symbol table information, and can optimize the pattern of kernel calls" [1]. Although Linda is not built on a truly object-oriented model, it provides abstractions such that processes act as "black boxes" receiving and sending messages [37]. Parallelism in Linda is often realized by the replication of parallel processes. Linda views a parallel program as a "spatially and temporally unordered bag of processes" [1], but not a process graph. Linda processes communicate implicitly by putting *tuples* into a virtual shared memory region called the *tuple space*. The *tuple space* may be implemented by using real shared memory or may be distributed across a network. A *tuple* is similar to a record, an ordered set of values, in relational databases. When a process wants to retrieve a data value, a matching operation is performed on the tuples in the tuple space. If a tuple in the tuple space matches the specification, it is retrieved and the process continues to execute. Otherwise, the process will be suspended until a matching tuple is put into the tuple space by another process.

Frameworks [44, 43, 45] is another tool which uses a model similar to the object-oriented model. An application graph is used to represent the structure of a program. The nodes in the application, representing computation units, are associated with different source modules written in sequential C. The sequential modules may also contain the additional *call* or *reply* keywords to indicate the locations of remote procedure calls and locations of waiting for replies from the calls. The application graph in FrameWorks differs from other similar dependency graphs in that the application may be restructured by attaching templates to the nodes. A node is typically composed of three separate templates: the input, output, and body template. The power of FrameWorks lies in the ease of converting sequential programs and restructuring parallelized programs. However, FrameWorks suffers from a user interface which is hard to use, and the need for additional keywords in the source modules. Because Enterprise is a new generation of FrameWorks, a detailed review of FrameWorks will be given in a later chapter.

PIE (Programming and Instrumentation Environment) [41] shares a similar template attachment approach with FrameWorks. An application in PIE also consists of a collection of functional modules. Templates may be used to encapsulate the functional modules to add new meanings to the modules. For example, a functional module may represent a part of a pipeline or a master-slave structure depending on the chosen template.

2.4 Other Issues in Parallel Programming

Besides the conceptual models described in the previous sections, there are other issues in the domain of parallel programming that affect the usefulness or the ease of use of different parallel programming tools.

2.4.1 Explicit and Implicit Parallelism

Explicit parallelism is accomplished by adding *explicit* calls or directives, by the user, into the source code of parallel programs. Examples of explicit parallelism are the *pardo loop* used on the Myrias SPS-2 computer [34], or the *call* and *reply* statements used in FrameWorks [44, 43, 45]. This approach makes the programmer know exactly how a program is being parallelized. The programmer is given complete control on the parallelization technique to be applied to the program. However, when the program is being restructured, minor modifications on the source code may be required.

On the other hand, implicit parallelism relieves the programmer from rewriting or editing his program even when the program is being restructured. For example, an optimizing compiler will try to parallelize user programs regardless of the structure of the program. However, the structure of the program has a large influence on the quality/performance of the compiled code. Yet another way to support implicit parallelism is to provide users a conceptual model on top of the hardware layer. By using the tools which facilitate the use of conceptual models, a user writes his program in terms of functionality and defines the parallel structure of his program by using a structured graph. The locations of the parallel code segments, such as *call* and *reply,* are already specified implicitly in the dependency graph. The tools, therefore, should be able to insert the calls into the source code without any help from the programmer. Thus, it is possible for this kind of tools to employ the implicit approach. Moreover, the implicit approach makes it even easier to convert a sequential program into a parallel program and vice versa.

2.4.2 Expressiveness

The expressiveness of a parallel programming tool decides what kinds of parallelism can be supported by such a tool. Although a tool may not be able to express all forms of parallel constructs, it may still be useful if it covers most of the commonly used constructs and is convenient to use. Except for languages like Strand 88 [22, 21] and some parallel programming environments such as Enterprise, most of the tools or models do not support the expression of recursive calls or dynamic use of divide-and-conquer parallelism.

2.4.3 Ease of Programming vs. Full Programming Control

The tools which are designed for parallel programming often provide to the programmer an abstract view of the underlying physical machine. This abstraction helps the programmer to concentrate on the parallel structure of his program rather than spending most of his efforts in dealing with hardware details. This ease of programming does not come without a cost. In the study of Jones and Schwartz [26], they point out that the masking of the hardware architecture in a distributed system often comes with a severe performance penalty. To achieve good performance in a parallel computing environment, the programmer must have some control over the mapping between processes and processors. Ideally, a parallel programming tool should provide both the options of automatic processor allocation for naive users and full processor allocation control for advanced users.

2.5 Conclusion

In this chapter we have looked at various issues in designing tools for developing parallel programs. One important issue is to provide the programmer with a stable and familiar programming environment. The environment should also encourage the user to write parallel programs in terms of a virtual machine. The advantage of using a virtual machine is the increased portability of the programs. Several models of virtual machines are available and are used in various parallel programming tools. The task-oriented model represents parallel processes as concurrent tasks. The data-oriented model views a program as a collection of computation nodes, and the nodes are activated by sending data to them. The newest available model is the object-based model. The object-based model seems to be the most powerful model among the three. Programs developed under the object-based model can be restructured easily and new objects may be implemented by using other basic objects.

Many of the recently available parallel programming tools try to provide the programmer with an integrated programming environment. Older parallel programming tools often only allow the programmer to perform one specific operation, such as analyzing or debugging a program, in the environment. The programmer must switch to another environment to perform other actions such as modifying the source code or compiling the program. The newer generation tools allow the programmer to edit, compile, execute, debug, and monitor a parallel program without switching from one environment to another environment. The integrated programming environment, therefore, provide a better and more intuitive environment for the programmer.

Chapter 3

The FrameWorks System

The FrameWorks system is a programming environment for developing distributed applications [44, 43, 45]. FrameWorks allows users to separate the code of their distributed applications and the specifications of their distributed structures. In FrameWorks, a user writes applications in terms of functional *modules*. Each of these modules represents a collection of sequential procedures and are enclosed by *templates*. The templates are used to hide the implementation details such as the communication and synchronization among other modules. This approach reduces the complexity of developing distributed applications significantly. The separation of the functional modules and their relation to other modules also makes the developed applications much easier to modify and to adapt to different processor constraints.

This chapter serves to provide an overall description and a review of FrameWorks. The system is of particular interest because it is the predecessor of Enterprise. Many useful ideas developed in FrameWorks have been incorporated and improved in Enterprise. In addition, some of the limitations in FrameWorks have either been eliminated or solved in Enterprise.

3.1 The FrameWorks Model

The FrameWorks model allows a user to write distributed applications in terms of functional modules, and delay the concern of issues such as synchronization to a later time. This separation is realized by employing the concept of ***template attachment***. An application in FrameWorks is separated into two parts. The first part can be represented by a high-level directed graph, called an ***application graph***. A node in the graph represents a computation unit, and an edge in the graph represents a communication path between two nodes. Two computation units may communicate only if there is an edge connecting the two units. The communication is handled by a way similar to the remote procedure call (RPC) protocol. The second part of an application includes the functional ***modules***. Nodes in the application graph are associated with functional modules. These modules represent collections of sequential procedures necessary to perform the computations. Each module is enclosed by ***templates*** to define its communication and synchronization properties. The implementation of these properties, however, is hidden from the user.

3.1.1 Modules

Modules in FrameWorks, called ***FW modules***, are collections of sequential procedures. Every FW module contains exactly one ***entry procedure*** which can be called by other modules in an application. Other procedures are treated as ***local procedures***, and can be called only within the module itself. During the execution of a FrameWorks application, every module (functional unit) is represented by an individual process. Therefore, the interaction between the modules represents the interaction of different processes running on different processors.

FW modules are similar to ordinary C modules. However, there are several major differences. All FW modules end with the ".f" suffix, while C modules end with the ".c" suffix. Furthermore, in a FW module, the entry procedure has to be the first procedure appearing in that particular module. The syntax of a FW module looks almost identical to a C module, except for two additional keywords, *call* and *reply*. The *call* statement is used to call an entry procedure, which resides in a different FW module and may be running on a different workstation on the network.

For example, if in an application module X calls another module Y, the call can be made in X by the statement:

```
call Y (input);
```

where *input* contains the parameters to be sent to Y. In FrameWorks, the parameters passed between different modules are called *frames.* A frame is a structured message, in the form of a C structure. It contains all the parameters to be passed to another module. In other words, each module call may contain only one arbitrarily complex parameter. Furthermore, the user of FrameWorks is responsible for explicitly packaging the parameters into frames.

Different from simple RPC calls, the calling module in FrameWorks may choose to continue its execution or to wait until a frame is returned from the called module. Asynchronous communication is assumed when the call statement is used in its basic form as described above. The calling module continues its execution without waiting for the called module to finish. When a return frame is expected from a module call, the

```
output = call Y (input);
```

statement may be used, with *output* being the frame to be returned from Y. In this case, the calling module (i.e. X) will be suspended until the output frame is received from Y. This is similar to the use of synchronous communication between the two modules. The *reply* statement,

```
reply (output);
```

is used in Y to return an output frame (answer) back to X, the calling module. Similar to the input frames, the user is responsible for packaging the return values into an output frame.

3.1.2 Templates

For a distributed application to function properly, it must handle the synchronization, communication, and scheduling of messages being passed between different processes. As described in the previous section, the only distributed information contained in FW modules appears in *call* and *reply* statements. Additional information is contained in *templates.* The templates are used to hide the low-level implementation details, such as how a message is sent to other modules, from the user. Up to three different kinds of templates may be attached to a FW module: an input, output, and optional body template. The input template specifies how incoming messages are handled by the module. The output template specifies the handling of outgoing messages. Commonly used structures in parallel programming, such as a pipeline or a master-slave relationship, can be specified by using different combinations of the input and output templates.

An optional *executive* body template may be used to modify the behavior of a FW module. The executive template is used to re-direct the input and output of an application to the user's terminal. If the template is not used, the application can be running in the background without user interaction.

There is also a *contractor* body template which may be attached to a FW module. The application graph used in FrameWorks is a static specification of the interaction between different modules. The number of concurrent processes is fixed and cannot be changed during execution time. However, the *contractor* body template may be used to change a FW module's behavior. This template tells the system to *contract out* work dynamically to different workstations according to the size of the problem. Therefore, when more computation power is needed, additional *employee* processes will be created to share the work if an idle workstation is available. When the additional computation power is no longer necessary, the employee processes will be removed by FrameWorks. This mechanism allows a FrameWorks application to adapt easily to the changing processor constraints in a workstation environment.

One important feature in FrameWorks is that the use of templates does not require any changes to the FW modules. The templates simply attach additional meanings to the FW modules. The use of the template attachment approach also encourages restructuring of a finished application. An application could be restructured to adapt it to the distributed environment or for fine-tuning performance.

3.2 Developing Applications using FrameWorks

In developing a distributed application using FrameWorks, two types of input are needed from the user. The first one is the source code modules (i.e. FW modules). The modules usually have few changes from their sequential versions. The user is required to modify the modules by adding the *call* and *reply* keywords at their proper locations, and changing the parameters of these calls into *frames*. It may also be necessary to reorganize the source code and to group the procedures into different FW modules.

After each of the FW modules is developed, FrameWorks requires the user to specify a communication graph between the modules. This graph describes how one module is related to the others in the program. The graph can be constructed by using the *Mod_Craft* program, a graphical interface in FrameWorks for creating communication graphs. After both of the FW modules and the communication graph are created, the user may compile the application by the commands *fwconfigure*, *fwpp* and *compile*. Details of these commands and the Mod_Craft program will be described in the following sections.

3.2.1 Mod_Craft: The Graphical User-Interface

Mod_Craft is a graphical user-interface in FrameWorks that allows users to specify the application graphs of their programs. The different input, output, and body templates are represented by icons in the interface. A graph is constructed by creating nodes and by drawing edges between the nodes. A node in the graph is formed by combining the icons of an input template and an output template together. Each node is associated with a FW module, and additional resource utilization constraints can be specified in the interface. For example, the user can specify that there should be three copies of a particular FW module running in parallel and none of them should be running on workstation X. The output of the interface is a file named as "application_name_templates." This file is a textual representation of the graph specified in Mod_Craft. The file will be used later by the commands fwconfigure, fwpp and compile to produce an executable version of the program.

The graphical user-interface of FrameWorks for Sun workstations is implemented by using Diction, Chisel and Vu [46, 47] which are collections of user interface management tools. These tools allows rapid prototyping of graphical user-interface under the SunView environment. The use of these tools shorten the development time of FrameWorks. However, these tools also limit the portability of the system, because they are not commonly supported in other computing environments.

3.2.2 Compiling FrameWorks Applications

While Mod_Craft can be considered as the front-end of FrameWorks, there is a back-end of the system. The back-end of the system is responsible for inserting the low-level distributed code, compiling the application, and setting up the runtime environment for the application. There are three commands used for accessing the back-end of FrameWorks to produce an executable distributed application. They are *fwconfigure*, *fwpp*, and *compile*. After the executable is created, the user can invoke the application by simply using the application name, and the distributed processes will be created automatically by the system.

The functions of each of these commands are listed as follows:

1. *fwconfigure* <application_name>: This command takes the output of Mod_Craft, the *application_name_templates* file, and outputs the following files:

   application_name_allocation: this file contains the specifications as to how many processes are assigned to a particular workstation. One entry is needed for every workstation used.

application_name_config: this file contains the information needed to start up the application. This file is actually the "config" file used by NMP [32], the low-level library used by FrameWorks to handle message passing.

application_name_congraph: this file contains the information on the input, output, and body templates of each FW module. The connections between the modules are included in this file.

application_name_list: this file contains the names of the FW modules. It is needed for "compiling" the application by using the *compile* command.

application_name_node_table: this file contains the names of every node in the application graph and an unique identification number is assigned to each node.

2. *fwpp* <application_name>: *fwpp* is actually a C pre-processor which takes the FW modules and translates them into normal C modules with the ".c" suffix. The translation from FW modules to C modules involves adding the code necessary for handling the *call* and *reply* statements. Since the code used for the two statements depends on the structure of the program, *fwpp* consults the *application_name_templates* file so that it may generate the distributed code accordingly.

3. *compile* <application_name>: This command compiles all the C modules generated by *fwpp* and produces a distributed program which can be run on a network of workstations. *"Compile"* knows what modules, by using the *application_name_list* file, are included in the program; thus the user does not have to construct a makefile for the program manually. Users of FrameWorks can specify the necessary libraries or compilation flags in a file called *fw_env* which has to be located in the same directory as the program itself.

3.2.3 Code Generation in FrameWorks

The back-end of FrameWorks is responsible for generating the code needed to handle the communication and synchronization between different processes (FW modules). The code currently generated are NMP calls [32]. NMP is a package of high-level interprocess communication routines designed for UNIX systems. Using NMP has the advantage that if a user is familiar with the package, he may further optimize the generated code without learning sockets. On the other hand, there is also a portability problem when NMP is used as the backbone of FrameWorks. If a user decides to port FrameWorks to another system, he has to port the NMP package onto that particular system first.

3.3 Debugging facility

The debugging facility in FrameWorks is provided by the program *View_Graph.* It is a post-execution analysis tool for FrameWorks applications. The tool is used to replay the execution of a program, through its *application graph,* on a color graphic terminal. The tool changes the color of the link between two modules to reflect the fact that a message is passed between the modules. Similarly, it changes the color of an icon in the graph to reflect the status of a process, such as a module is waiting for an input frame to arrive. The animation of the program execution could be used to detect deadlock conditions and potential performance bottlenecks. The user is allowed to step forward or backward through the program animation. Besides the graphic display, there is also a *status* command in the tool which allows users to query the current status of a FW module. The command can be asked to return performance statistics or to display program trace information. View_Graph is still in its preliminary stage when the development of FrameWorks was stopped, but it provides a valuable means for debugging FrameWorks applications.

3.4 Communications Between Different Components

In the current implementation, communication between different components of FrameWorks is mainly through files. This approach has the drawback that most communications between the modules are virtually one way channels. Because of these channels, FrameWorks cannot provide dynamic information back to the users. In other words, these channels limit the functionality of FrameWorks, and provide little feedback to the users of the system. Currently, the user-interface only allows users to specify the communication graph of an application. Users must leave the FrameWorks environment to make changes to their source code, or to compile their applications.

FrameWorks generates many small files such that one component may interact with other components in the system. These small files not only clutter the directory in which an application resides, but also confuse the users as they look into the directory. If these files are either hidden or grouped together into a bigger file, it should be easier for a user to extract important information by simply looking into the directory itself.

3.5 Limitations of FrameWorks

Although the FrameWorks system provides users a better programming environment for writing distributed applications, FrameWorks has several limitations in its own right. The current version of FrameWorks supports only a homogeneous network of workstations. This is usually not the case in a real life computing environment. It is very common in workstation environments for a network to contain several different kinds of workstations which are not binary compatible. Another major limitation of the FrameWorks system is the fact that the communication graph is static, with the exception of when the *contractor* body template is used. This restriction makes the creation of dynamic processes limited to the use of the *contractor* body template. However, because of the limitations of NMP, the full functionality of the template is restricted.

There are also other limitations in the FrameWorks system. These limitations do not directly affect the functionality of the system, but merely make the system less convenient to use. One of these is that an entry procedure must appear as the first procedure in a FW module. FrameWorks also imposes a strict naming convention on the FW modules. A FW module must be named as its entry procedure name ended with the ".f" suffix. However, the ".f" suffix is often being used to name Fortran code modules. Finally the icons used in drawing the application graphs and the lack of textual labels make the distributed structure of a FrameWorks application less easier to understand.

Enterprise was developed to eliminate the many limitations of FrameWorks. Enterprise represents an evolution of FrameWorks, and uses many ideas developed in the earlier system. For example, the concept of *template attachment* is also used in Enterprise. Programming in Enterprise, however, provides a higher level of abstraction for the user. The application graph in Enterprise is also easier to construct and understand when compared to FrameWorks. Application graphs in Enterprise show not only the communication links between the modules, but also represent the high-level parallelization techniques graphically. In an Enterprise graph, a node is a single entity containing a combination of the input, output, and body templates. The user is not required to draw any links in constructing the graph. To represent different commonly used parallel programming structures in the graph, the user changes the icon by either *coercing* or *expanding* it using the graphical interface. In fact, the icons in Enterprise contain the set of all legal combinations of input, output and body templates in FrameWorks. The Enterprise icons also support a hierarchical structure in a way that a single icon may be used to represent another subgraph and to avoid presenting too much information to the user.

Another major difference separating Enterprise from FrameWorks is the removal of additional keywords from the sequential programming language. Enterprise uses the information in the application graph and a compiler to insert the low-level distributed code at the proper locations. The removal of the keywords has two major effects. First, a functional module is now exactly the same as its sequential version. Therefore, to parallelize an existing application, the only actions required from the user are the regrouping of procedures into different functional modules and the construction of the application graph. To restructure an Enterprise application, the user is not required to insert or to remove keywords from the functional modules. Second, because there are no additional keywords, Enterprise modules can be compiled for sequential execution with no or little changes in the source code. In other words, the conversion between the sequential or the parallel version of an application can be performed easily. In general, Enterprise offers a better programming environment over FrameWorks. The major strength of Enterprise lies in its simplicity of use. This simplicity not only allows programs to be constructed more rapidly, but also reduces the possibilities of user errors.

3.6 Conclusion

The FrameWorks system was one of the earliest systems designed for providing an easy approach to writing distributed programs for a network of workstations. It achieves the goal by separating the source code of the distributed program from its underlying distributed structure. This allows a program to be restructured easily for different parallelization techniques or for different processor constraints. The implementation details of low-level communication protocols are handled by using *template attachment*. This approach facilitates the writing of distributed applications or the parallelization of existing applications. On the other hand, FrameWorks has several limitations at its current status. These limitations restrict the overall usefulness of the system. As a result, the system was redesigned and re-developed into a new system, called Enterprise. While only some of the major differences between the two systems have been outlined to this point, a detailed description of Enterprise will be presented in the next chapter.

Chapter 4

The Enterprise System

4.1 Introduction

In Enterprise, a parallel/distributed program is organized like a sequential program. The structure of a program does not depend on whether it is designed for sequential or distributed execution. This allows a compiler to be used to compile the program if sequential execution is desired, or Enterprise to be used for distributed execution. The easy conversion between a sequential and a distributed program is possible because none of the distributed information of an application is stored in its source code. Any communication or synchronization code for distributed process interactions is removed from the user's responsibility. Instead, the information is specified in an *asset graph*, which is similar to a control-flow graph in other parallel programming tools. Enterprise then automatically generates the necessary code to handle the process interactions.

An Enterprise program consists of a collection of sequentially executed modules. Parallelism in an application program is introduced by allowing the modules (communicating processes) to run concurrently on different workstations. Each of these modules has a single *entry procedure* and a set of *internal procedures*. The entry procedure is the only procedure which is visible to and may be called by external modules. A module may contain any number of internal procedures, but they can only be accessed locally within the module itself.

There are no common variables allowed among Enterprise modules. The communication between the modules is handled by a modified form of remote procedure calls. The user views a remote procedure call just as any normal procedure call, and does not worry about the distributed nature of the call. Enterprise generates the communication code that is needed for translating it into a distributed call, while the implementation details are hidden from the user. At the lowest level, Enterprise calls are translated into messages passed among modules. However, the user can specify the way that a module interacts with other modules.

The interaction among modules is specified by the *role* of a module and the *call* to the module. To specify the role of a module, the user selects one of the parallelization techniques (asset kinds) available in Enterprise to be used when the module is invoked. The call to a module, similar to a procedure call, defines the identity of the called module, the information (arguments) to be passed, and the information (return values) to be returned. The role of a module is similar to a template in FrameWorks, and is specified graphically in an asset graph. The call to a module is specified as a procedure/function call in the source code.

The following sections will be used to describe the programming model used in Enterprise and the overall architecture of the system. Section 4.2 describes the module calls, section 4.3 describes the roles (asset types) of a module, and section 4.4 describes the properties of an asset graph of an Enterprise program. Section 4.5 describes the overall architecture of Enterprise. The implementation of part of the system will be presented in section 4.6.

4.2 Module Calls

Enterprise augments the C programming language with new semantics, rather than new syntax, for function and procedure calls. Since Enterprise does not use additional keywords, such as the *call* and *reply* statements used in FrameWorks, an Enterprise program has the identical syntax of a sequential C program. Enterprise module calls and normal function calls are syntactically indistinguishable. Enterprise uses the information stored in the asset graph to differentiate the two types of calls. In sequential programming, a function call is a subroutine call which returns a result, and a procedure call is a subroutine call which does not return a result explicitly. A similar distinction is made in Enterprise, where a *f-call* refers to a module call that returns a result back to its caller, and a *p-call* refers to the one that does not.

Although an Enterprise call and a sequential call share the same syntax, they differ in the following ways:

1. Pointer type parameters, or structures which contain pointers, are not allowed to be used as arguments in Enterprise calls. Therefore, in the C language version of Enterprise, a result can only be returned through the use of a *f-call,* but not by side effects (because the parameters of a function call in C are passed by value).

2. The value returned by an *f-call* must not be a pointer type. In other words, the return value must be a variable of a predefined type or a C structure with a fixed size.

3. A module call in Enterprise does not suspend the calling module. For example, if module A makes a call to module B, module A continues to execute. However, if the call is a *f-call,* and module B has not finished its computation, module A would suspend itself when it tries to use the return value.

Because of the syntax used by an Enterprise call, the task of transforming sequential programs into parallel ones or of changing the parallelization techniques being used becomes trivial. The user transforms a sequential program by breaking it into smaller modules, and by associating the parallelization technique used by each module through a graphical user-interface. In general, the selection or the change of the desired parallelization techniques associated with an module may require only minor or no modification on the original source code.

The *f-call* in Enterprise is not necessarily blocking. Instead, the calling modules blocks only if the result is needed and the called module has not yet returned. Consider module A containing the following code segment:

```
result = B (data);
/* Some other code */
answer = result + 1;
```

If the "some other code" section does not use the value of "result", module A may continue its execution without any effect on the correctness of the program. Therefore, the calling module A only has to block when the statement "answer = result + 1" is reached, and the called module B has not returned the value of "result". This concept allows more parallelism to be introduced into Enterprise programs. However, this idea is not totally new, but is similar to the work on *futures* in object-oriented programming [14]. In the case that module A makes a *p-call* to module B, as in the statement

```
B (data);
```

the call is non-blocking, so that both A and B may continue to execute concurrently. Again, the concurrent execution of the two modules does not affect the correctness of the program, because there are no common variables allowed among modules. However, the use of the delayed blocking creates some semantic ambiguities in some programs. These problems will be addressed in a later chapter which describes the implementation of the compiler used in Enterprise.

4.3 Module Roles and Assets

The role of a module defines the parallelization technique used and is independent of its call. A fixed number of predefined roles, represented by different *asset* kinds, are available in Enterprise. The use of the term "assets" corresponds to the analogy of how work is divided, distributed, and executed in an organization. This analogy was chosen because of the inherently parallel and hierarchical structure of an organization. Moreover, the names used for describing the module roles will be easier for new users to understand.

An organization usually has various resources, or *assets,* to perform its assigned work. The work is often divided into smaller sub-tasks, and these smaller tasks are then distributed to different parts of the organization, such as departments, divisions or individual workers, to work in parallel. In the case where the organization has no direct concern on the number or the nature of individual workers who perform the tasks, it contracts out the tasks to different contractors. The contractor may utilize a suitable number of resources, which depends on the size of the task, to fulfill the contract. Besides the different assets, an organization often has some standard shared services (for example, time keeping, information storage, and equipment supplies) available to its workers for completing the tasks.

Currently, Enterprise supports seven asset types: individual, line, department, pool, contract, division, and service. The following sub-sections will be used to describe the properties of the assets.

4.3.1 Individual

An *individual* asset is similar to an individual worker in an organization. This asset contains no other assets, and it represents a sequential process in an application. Therefore, a sequential program can be constructed by using a single individual asset, and whose entry procedure is *main()*. When an individual is called, it handles the call sequentially to completion. Any subsequent call to the same individual cannot be started until the previous call is finished. An individual is visible to external assets as a normal procedure and may be called by using its name.

4.3.2 Line

A *line* asset is similar to an assembly line which contains a fixed number of stages or stations. Each station is responsible for refining the work of the previous station. For example, the making of a car may be divided into three stages: the first stage builds the frame of the car, the second stage adds in mechanical parts, and the last stage adds the interior and finishes the body work. A line in Enterprise contains a fixed number of heterogeneous assets in a fixed order. Each asset calls the next asset in the line using module calls. The first entry in the line represents a *receptionist*, who receives work form external assets, and sends the work to the first asset in the line. Subsequent calls to the line wait only until the first asset finishes its work, but not until the entire line is finished. A line is commonly referred to as a pipeline in the literature.

4.3.3 Pool

A *pool* asset is similar to a pool of workers, such as a group of bank tellers, in an organization. Each member of the pool performs an identical task. In Enterprise, a pool contains a fixed number of identical assets. Every asset shares the same asset name and the common code module. Since the pool members are externally indistinguishable and share the same name, a call made to a pool asset cannot select a specific pool member to perform the task. When a call is made to a pool, an idle asset executes the call. However, if all member assets are busy, then the call waits for one of the assets to finish. A pool is analogous to a master-slave relationship, where the number of slaves is fixed at compile time.

4.3.4 Contract

A *contract* in Enterprise is similar to an agreement used in building construction firms or that used in courier companies. Although a pool of workers could be used to handle the assigned work, the use of a pool requires the organization to allocate a number of workers to finish the work. The number of workers is defined statically in the structure of an organization, and cannot be adjusted for different problem sizes. A contract relieves the organization from specifying the exact number of workers required by the work, and allows the contracted company to allocate a variable number of workers to accomplish the work according to the work load. Additional workers may be hired when the load increases, or existing workers may be laid off when the load decreases. A contract would be useful when there is only a limited amount of resources in the organization, and the amount of work involved in a particular task has large variances. For example, when an organization has a task to be finished, such as the delivery of merchandise, the contracted company is informed and it may use as many resources as needed to complete the contract. The delivery time could be affected by the amount of merchandise to be delivered and the number of resources available to the contracted company. Similarly, the execution time of a contract in Enterprise depends on both the size of the task and the amount of resources available to the contract.

A contract asset is similar to a pool asset in that each member asset has the same name and the same code. However, the number of assets contained in a contract is dynamic. This number depends on the amount of parallelism involved and the number of idle workstations currently available on the network. When an Enterprise call is made to a contract, an idle member asset executes the call. If all assets in the contract are busy, then the call cannot be started until one of the assets finishes its work. However, since the number of assets contained in a contract is dynamic, a new asset may be added to the contract to handle new calls. A new member asset is added when all assets are busy and an idle workstation becomes available in the network. A contract is equivalent to a dynamic master-slave construct, where the number of slaves varies in response to the size of the problem and the amount of available resources.

4.3.5 Department

A *department* is an asset which contains a fixed number of heterogeneous member assets. The tasks assigned to a department are directed to the member assets to work in parallel. A department asset in Enterprise is different from a line asset in which a member asset does not call other assets in the department directly. The member assets can only be called by the receptionist of the department. The receptionist is the only asset which is visible and is callable by external assets. It is responsible for receiving work from other assets, and for assigning the work to the appropriate member asset in the department. A department is similar to the *fork* and *join* construct commonly found in some programming languages. Member assets in a department are not restricted to being individual assets. Each member can assume any predefined role in Enterprise, such as a line, a contract, or a department.

4.3.6 Division

A *division* represents a collection of hierarchically organized identical assets. These assets are organized in a tree-like structure with a fixed breadth and depth. A receptionist, who shares the same name of the division, is responsible for receiving work from external assets. The received work is divided and distributed at each level of the division. Unlike other assets, a division may call an instance of itself recursively. A division is used to parallelize divide and conquer computations. The maximum level of distributed recursion, however, is set at compile time because the user specifies the fixed breadth and depth. When the maximum depth of recursion is reached, local procedure calls rather than remote procedure calls are used to finish the computation. The first reason of using local procedure calls is to reduce the overhead involved in sending messages across a network. As the recursion goes deeper, the granularity of the divided work may become to small to be benefited from additional distributed processing. The second reason is due to the limited number of workstations available on a network. A division generally contains a lot more workers (computation nodes) than the number of available workstations. Therefore, if only remote procedure calls are used, multiple Enterprise processes will be put on each workstation creating unnecessary context switching activities and inter-process communication overheads. Local procedure calls allow the workstations to be used in a more efficient manner.

4.3.7 Service

A *service* in Enterprise contains no other assets. It is similar to an individual asset, but it may be accessed by more than one asset in parallel. A service has the special properties that it is not consumed by use, and the order of use does not affect the correctness of a program. A notice board in an organization can be considered as a service. Each individual worker may look at the notice board to obtain the latest information about the organization. A service is designed to be accessible by any asset in an application. This makes it possible for other assets to access the global state of the program at any time, or even the implementation of a virtual global memory across the network.

4.3.8 Other Asset Kinds

In the current design of Enterprise, a pool asset is designed to be a *synchronous* pool; results produced by Enterprise calls must be collected in a *fixed* and *ordered* fashion. The order of the results being collected is specified in the user written code, and is fixed at compile time. An example of receiving the results in a fixed order is by using a *for-loop* to collect the outstanding results. However, this may not be the desirable behavior in some Enterprise programs. If each Enterprise call involves a variable amount of work, the collection schedule specified by the user may hinder the overall performance of an application program. In the example above, the program must wait for the first result to return, before the rest of the results can be used (received). If the first result being collected is the one that requires the longest computation time, a large amount of waiting time would be spent needlessly.

The *asynchronous* pool is designed to solve the above scheduling problem. An asynchronous pool allows outstanding return values to be collected in an unordered fashion. When a return value is needed, an unordered bag is searched for any valid values. If such a value can be found, the value will be used by the program; otherwise, the program will wait until any valid value is returned. In other words, the program will spend less time waiting for replies, and spend more time doing computations. The exact semantics of an asynchronous pool is still under construction.

Another asset which is under construction is the *dynamic* division. The current implementation of a division has a fixed breadth and depth. The advantages and disadvantages of using a division which has a fixed breadth but a dynamic depth are currently being investigated. The depth of a dynamic division varies according to the task size and the number of idle workstations available in the network. A static division is similar to a pool which may call itself recursively, and a dynamic division is similar to a contract which may call itself recursively.

4.4 Enterprise Graphs and Parallelism

Enterprise allows the user to construct an asset graph using a simple graphical tool. There is a significant difference between the graph used in Enterprise and the dataflow graphs used in many other parallel programming tools. Other tools, in general, require the user to construct a graph by drawing nodes, and by using edges to join the nodes in the graph. The nodes in the graph represent processes, and the edges represent communication channels. The asset graph used in Enterprise is constructed in a novel way. A graph starts with a single individual asset, and the rest of the graph is constructed by changing the role of the asset to introduce parallelism into the program. For example, an individual may be *coerced* into a line asset with three members. The members can be *coerced* further into other asset kinds. The asset graph in Enterprise has the following properties:

1. Each asset (icon) in the graph represents a high-level parallel construct. For example, the construct can be a line, a pool, or a contract. These constructs allow a graph to be constructed at a higher level of abstraction, and ensure that the communication code will be generated correctly.

2. The user is relieved from drawing the nodes and connecting the nodes by edges. The dataflow graph used in other parallel programming tools are usually dense, and sometimes multiple edges are needed to connect two nodes together. The approach used in Enterprise results in a graph that is easy to understand, and reduces drawing errors in constructing the graph.

3. The inherent hierarchical structure of an asset graph allows the user to look at the graph at different level of abstraction. An asset in the graph can be *expanded* or *collapsed* to control the level of abstraction and to manage the complexity of the program.

4. The asset graph shows a clear picture of data flow and parallelism in a program. The flow of information (data) of a program is expressed from top to bottom, and parallelism is expressed from left to right. In other words, the length of the graph reflects the critical

path of an application, and the width reflects the degree of parallelism.

5. The parallelization technique used for a module can be easily changed through the asset graph, because the technique is specified graphically and is independent of the code.

4.5 Enterprise Program Construction

A distributed application in Enterprise consists of two separate components: an asset graph and source code modules. The asset graph is created by editing icons through a graphical user-interface, and the source code modules are written in the C programming language. In this section, we will illustrate the construction of an Enterprise program through an *Animation* example.

The *Animation* program was contributed by a research group in our Department. This program is used to model and animate the behavior of a school of fish swimming across a display screen. There are three major operations in the program: *Model, PolyConv,* and *Split.* The functions of the operations are:

Model: Generates a frame for the animation by computing the location and the motion of each object in the display according to its behavior. *Model* stores the created frame in a disk file, calls *PolyConv* to process the frame, and proceeds to the next frame.

PolyConv: Reads in a frame created by *Model*, performs graphical transformations such as data transformations, viewing transformations, and projection transformations. *PolyConv* then calls *Split,* passing to it a transformed frame and a sequence number, to render the final image.

Split: Performs the final refinements on the transformed frame by doing hidden surface removal and anti-aliasing, and outputs the rendered image in a file.

The animation program can be parallelized by separating it into three different Enterprise modules: *Model, PolyConv,* and *Split.* These modules are simply created by selecting and grouping the appropriate functions into the right modules. The user then creates an application graph to describe the relations between the different modules. Finally, the information in the application graph and the source modules are used by Enterprise to generate the necessary low-level communication code, producing a distributed animation program.

In this example, *Model* does not need to wait for *PolyConv* to finish before it can start computing the next frame. Similarly, *PolyConv* does not need to wait for *Split*. Furthermore, *Split* is responsible for most of the computation-intensive work for the program. Because of these observations, a *line* asset can be used to structure the program. The last member of the *line* is coerced to a *contract* such that more workstations may be used to share the computation-intensive work. Figure 4.1 shows an Enterprise window, consisting of a canvas containing the asset graph and an asset palette containing one icon for each asset kind. The

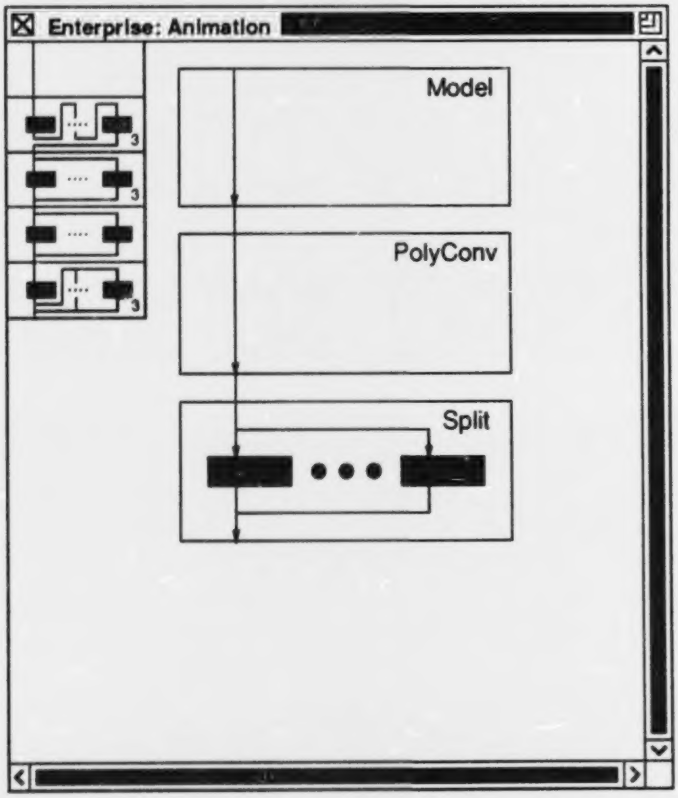

Figure 4.1: A Line with a Contract

graph actually shows a line of three members with the first two being *individuals*, and the last one being a *contract*. The asset graph also has a textual representation that is read by other components in Enterprise and the textual graph is shown as follows:

```
line 3 Model PolyConv Split
contract Split
Model
      library -lUTILITY -lfb -lm
PolyConv
      library -lUTILITY -lfb -lm
Split
      library -lUTILITY -lfb -lm
CFLAGS = -f68881
```

For the asset graph shown above, at least three processes will be running in parallel when the program starts up: one for *Model*, one for *PolyConv*, and at least one for *Split*. As the program runs and processors become available, the number of *Split* processes will grow dynamically. Enterprise also allows the user to restructure a program without modifying the source modules. For example, the user may coerce the *contract* in Figure 4.1 into a *pool* of two workers as shown in Figure 4.2. In this case, exactly four processes will be used by the program: one for *Model*, one for *PolyConv*, and two for *Split*. This example shows only some of the possible schemes that can be used to parallelize the *Animation* program. Of course, other asset graphs can also be used to structure the program to adapt it to the network of workstations.

4.6 The Architecture of Enterprise

In Enterprise, the analogy of an organization is used in the structuring of application programs as well as in the design of the system itself. The architecture of Enterprise consists of six logical components: an interface manager, an application manager, a code librarian, an execution manager, a monitoring/debugging manager, and a resource secretary, as shown in Figure 4.3.

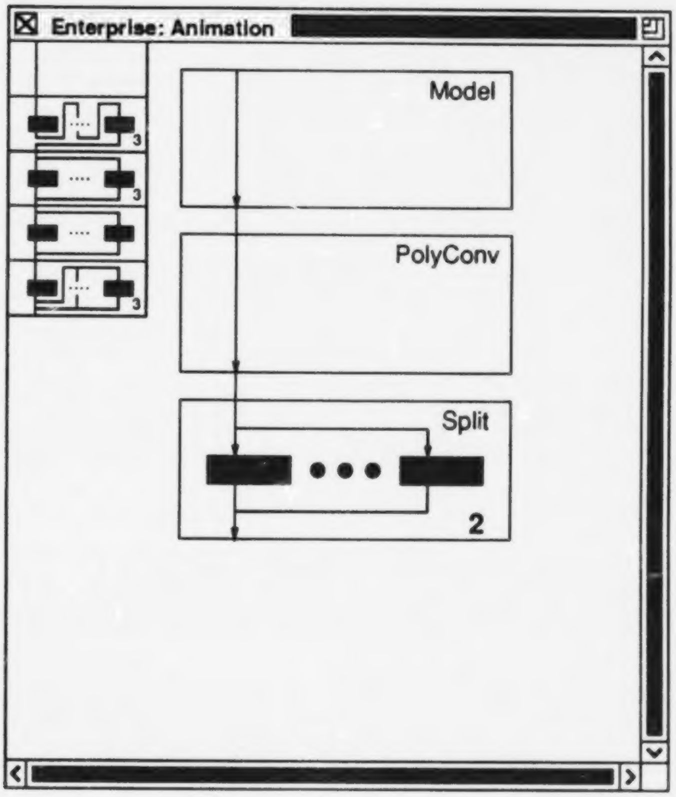

Figure 4.2: Coercing a Contract to a Pool

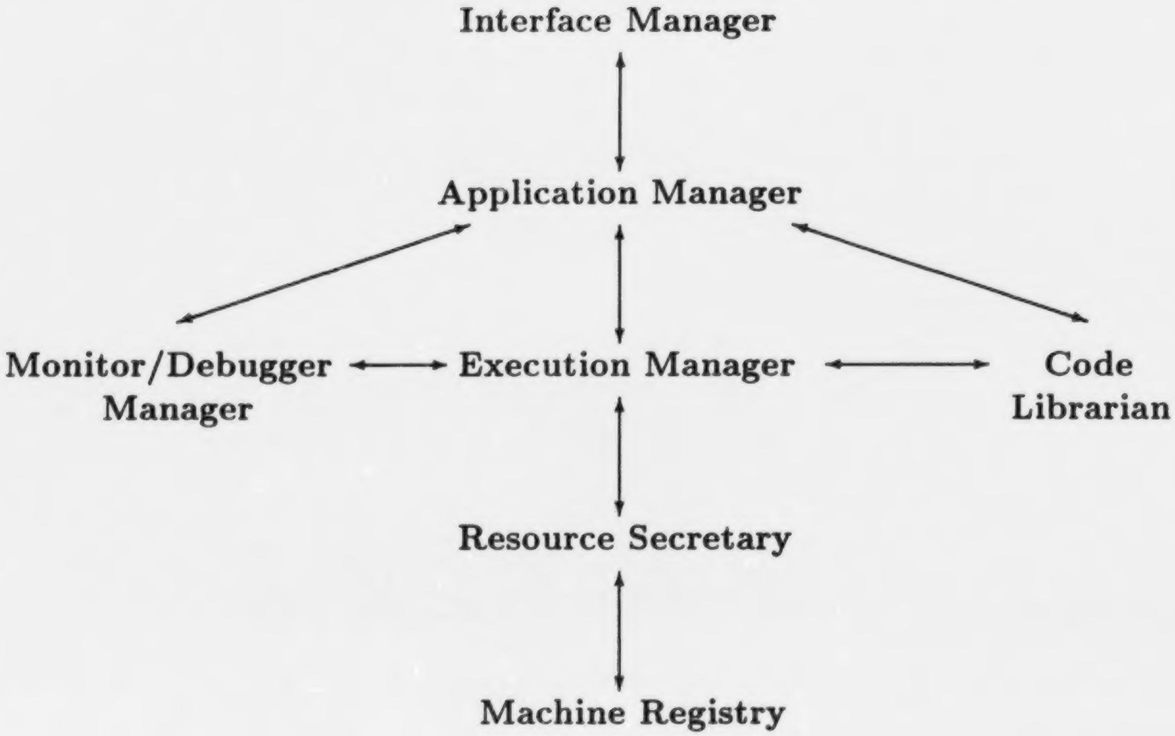

Figure 4.3: The Architecture of Enterprise

4.6.1 Interface Manager

The interface manager is responsible for managing the graphical user-interface used in Enterprise. This interface provides an environment for editing, configuring, compiling, executing, monitoring, and debugging applications. It allows the user to develop and maintain parallel programs in a single unified programming environment. The interface manager allows the user to access the functions provided by other logical components in the system. An asset graph editor is also included in the tool for users to develop their applications using the organization analogy.

The asset graph editor uses the object-oriented concept to manage the assets in the graph. Each asset is an instance of an asset class, and is responsible for managing its name, attributes (such as the length of a line asset), components (such as the components of a line), source code, drawing itself, and expanding itself, etc. Inheritance is used extensively in managing the assets because many responsibilities, such as redrawing itself, are shared among different asset classes. Similarly, the other interface components, including windows, menus, and dialogues, are also objects in the editor.

The current user interface was implemented under the X window system (Scheifler and Gettys 1986) on Sun workstations. This increases the portability of the interface to different workstations. The implementation is written in C++ using the Motif libraries. The combination is chosen for easy integration to the rest of the system (written in C), for greater portability and for minimizing the overall development time of the interface.

4.6.2 Application Manager

The application manager is the control center for Enterprise. All of the information about a parallel application is maintained by the application manager. This includes the asset graph, locations of source and object files, runtime status, and performance statistics of an application. Application specific information can only be accessed through the application manager. In other words, the application manager also acts as a communication channel for the different components in Enterprise.

4.6.3 Code Librarian

The role of the code librarian is to manage the source and object code of the different functional modules in a parallel application. Since Enterprise is designed with networks of heterogeneous workstations in mind, the code librarian may need to maintain multiple object codes intended for various architectures. The code librarian includes a pre-processor which transforms an otherwise sequential program into a parallel program, and a makefile creation utility which creates makefiles according to the asset graph of a program. The makefiles can be parameterized to support the different target machines.

The librarian has to know the locations of the source code and corresponding object files for a particular function module. When an Enterprise application is about to started, the librarian determines whether all of the required executables are available, or if it has to perform some compilations for the user. The compilation of a module requires the information stored in the asset graph, which is available from the application manager, to determine the role of the asset and the code to insert. To compile an asset, the librarian uses a pre-processor to insert the appropriate Enterprise (parallel) code into the module according to the role of the module. Re-compilation of an asset is only necessary when either the code of the asset has been changed, the asset has been *coerced* to a different role, or when the asset is to be run on a machine for which an executable is not yet available.

Compilation requests may come from two sources: either from the user or from the execution manager. First, the user could request a compilation through the application manager, to reveal syntax or semantic errors of a program. Second, the execution manager can request an executable on demand at runtime. The execution manager could request an executable when a machine becomes available in the network. If the executable is not available, the execution manager decides whether to request the compilation or to use an alternate machine.

4.6.4 Execution Manager

The execution of an Enterprise application is managed by the execution manager. It is responsible for creating remote processes, controlling runtime behavior, and setting up the required communication channels between the processes. Since the decision on where a process should be executed is determined by the execution manager at runtime, the source code does not contain any machine specific information.

Although the execution manager determines where a process should be executed, a user can associate a *machine preference list* with an asset. The list specifies the selection criteria in choosing a suitable machine for executing a process. The user can select the desirable machines by using the names of the machines, the processing speed of the machines, the amount of memory in the machines, or by other physical properties. If a machine preference list is not used, an asset may be executed on any machine in the network. The technique used in performing the machine selections is similar to the one used in FrameWorks.

Choosing the appropriate workstation for running the remote processes is an important duty of the execution manager, especially in the case of a *contract,* where hiring or removing additional workstations may occur frequently. In a heterogeneous network, some workstations have a higher processing power than other workstations. It is desirable for the execution manager to choose the fastest available machine on the network. It is also desirable to migrate processes from heavily loaded machines to less busy machines. However, the cost involved in migrating a process from one machine to another could be expensive. For example, if a faster machine becomes available, a re-compilation may be required to migrate the process. In general, it is not easy to estimate the cost and benefit resulting from migrating a process from a slower machine to a faster one, or from a heavily load machine to a less busy one. These are issues for which we still do not have good answers.

4.6.5 Resource Secretary

The resource secretary provides a list of available machines to the execution manager in a *machine registry.* The machine registry contains a list of specifications for the machines in the network. This list includes the machines' physical properties, such as the architecture and processor type, the operating system, the amount of physical memory, and the latest load averages of the machines. The load average information is used by the execution manager to make decisions on where a process should be started, or where a process should be migrated.

In addition, every machine which can be used by Enterprise also has a configuration file. This file can be used by the owner of the machine to set up user privileges. For example, a particular machine may not be used by others, except the owner, during office hours. If no special restrictions are set, the machine can be used by any users at any time.

4.6.6 Monitor/Debugger Manager

Because of the complexity of parallel programs, the monitoring and debugging facilities are very important in Enterprise. These facilities allow the user to discover performance bottlenecks and potential synchronization errors by monitoring the execution of a program. The monitoring of a program is done by program animation. When a message is being passed from one asset to another, or when an asset is changed from idle to busy, these activities are time stamped and may be sent to the user-interface or logged to a file. The collected events can be used to display the execution of a program dynamically, or to replay the execution at a later time. The debugging facility in Enterprise allows the user to step through the animation of a program's execution by setting break points at the message level, and allows the user to examine the contents of the trapped messages.

4.7 System Implementation

Enterprise uses the ISIS package [7] to handle the invocation and communication between different processes. ISIS is a package of high-level library routines for handling processes creation, termination, communication, synchronization, and fault tolerance in a heterogeneous network of workstations. In ISIS, processes are grouped into different *process groups*. A process group may be formed by just a single process, but usually a process group is formed by a number of remote processes in the network. Messages are passed by broadcasting to a process group, and collecting replies in the same call. The broadcast of a message can either be blocking or non-blocking. A broadcast that is blocking suspends the calling process until a reply is received. A non-blocking broadcast forks off a child task which is responsible for waiting for the replies. The calling process may continue to execute, and to interact later with the child process to collect the results.

The pre-processor used to insert the appropriate code into the module source is actually a modified GNU C compiler [48]. A compiler is needed because Enterprise has to distinguish a *f-call* from a *p-call* through the syntax of a program. Enterprise also generates warnings when a *f-call* is made to a module which does not return a result. If such a situation occurs, the program could deadlock because a module is waiting for an event which will never happen. A compiler is also required because Enterprise needs to know the types of the variables being passed to package them into a corresponding ISIS message. A user no longer has to package the variables into messages (frames) as required in FrameWorks.

The executable produced by Enterprise, and part of the Enterprise system itself, are implemented as ISIS programs. The logical components described in the previous subsections can be considered as communicating processes. However, for efficiency reasons, only some of the components communicate through message passing. When message passing is not used, the processes communicates though the application manager to obtain the necessary information.

4.8 Conclusion

The programming model used in Enterprise has been presented in this chapter. The model is described by using the organization analogy because of the inherent parallel and hierarchical structure of an organization and the familiar names used in describing it. An Enterprise program consists of functional modules and an asset graph. The functional modules contain module calls which use the same syntax of normal function calls. A module call can either be a *p-call* or a *f-call*. A *p-call* allows concurrent execution of both the called and the calling modules. A *f-call* allows current execution of the two modules until the calling module uses the return value of the called module, and the called module has not finished. An asset graph is used to describe the asset kinds of the modules. The asset kind of a module specifies the parallelization technique employed. Currently seven asset kinds, representing commonly used parallel constructs, are supported in Enterprise. These asset kinds and their semantics were discussed in this chapter. Finally, the overall architecture of Enterprise, and the tools used in the implementation of the system were also presented. The subsequent chapters will be used to discuss in detail the actual implementation of the Enterprise librarian, and to discuss some of the semantic issues raised in the implementation of the Enterprise model.

Chapter 5

Source Code and Object Code Librarian

5.1 Introduction

The source and object code librarian is a separate module in Enterprise designed to ease the user's task of managing Enterprise applications. An Enterprise application typically consists of a set of individual assets. Each of these assets represents a process which could be run on a single workstation and could communicate with other assets through the inserted ISIS code.

Since Enterprise applications are designed to be run on a heterogeneous network of workstations, several copies of the object files or executable files of the same asset may have to co-exist at the same time. Typically, there is one executable file for each of the supported architectures. The executable copies of an asset differ not only in the compiled code of the module itself, but also in the libraries that are linked in. Furthermore, because we cannot overload filenames in the UNIX environment, we need to find a way to distinguish the different executable copies of a single asset and a way to manipulate the *Makefile* used to manage these files. Finally, the pathnames of the libraries used by different architectures may be organized in different ways. In general, one makefile is needed for each supported architecture to reflect these differences. To generate generic makefile templates for different architectures, *Imake* [19] is used as an implementation tool to create the *Makefiles*.

5.2 Management of Source and Object Code

There are many possible schemes which can be used for managing the source and object code of an Enterprise application. In the domain of sequential programming, the management of source and object code of an application is often done by using the *make* [20] utility. A *Makefile* is used to specify the dependencies of different object files, and thus provide a way to *re-make* or *re-compile* the application upon the user's request. The source and object code management of an Enterprise application, however, is more complicated in several aspects.

First, an Enterprise application is designed to run on different architectures in parallel. To support this feature, different executable copies of the same asset may have to co-exist during the execution of the Enterprise application. A mechanism for distinguishing among these different executable copies of an asset has to be provided.

Second, the intermediate object files (i.e. the ".o" files) of an asset will have to be distinguished as well. Many programmers choose to keep the intermediate object files around to reduce the overall compilation time involved in *re-making* a program. For example, a large program may be divided into a number of smaller modules. These modules are then compiled separately to produce the intermediate object files. When a re-compilation is needed, and the object files are not deleted, only the modules that were recently modified have to be re-compiled to reflect the changes in the corresponding object files. The new object files may then be linked with the older unaltered object files to create the new program.

The convention for naming these object files in sequential programming is to use the same name of the source module ended with different extensions. For example, the object file of the module "Model.c" will be named as "Model.o". In an Enterprise application, however, the intermediate object files needed to build an asset may in fact be compiled for different architectures. This situation happens easily if some of the object files or source modules are used by more than one asset. Therefore, it may be necessary to re-compile a source file to generate a new object file, even if the corresponding source was not modified.

Although the *file* command in UNIX may be used to find out the intended architecture of an object file, this method is neither an efficient nor a reliable one. Its ineffectiveness is due to the fact that the command must be repeated for every object file in the directory before they can be linked together. It is also unreliable because the *file* command returns the CPU type rather that the actual architecture that the object code is compiled for. In some cases, the *file* command may also return the wrong file type. For example, when the command is used to check the file type of a SPARC executable, the file type is determined correctly on a Sun SPARC workstation. However, when the same file is checked on a Silicon Graphic machine, the file type is incorrectly determined to be a data file. Due to these limitations, alternate methods of naming the multiple executable copies of an asset will be discussed in the next section.

5.2.1 Using Architecture Keywords

To solve the naming problem, two alternatives were considered. The first was to distinguish the object and the executable files by inserting appropriate architecture keywords into the filenames. For example, the executable of the asset PolyConv may be represented by PolyConv.sun3 and PolyConv.sun4, where "sun3" and "sun4" represent the architectures which the asset can be run on. However, this scheme of managing the source and object code of Enterprise assets suffers from many disadvantages.

One disadvantage is due to the increased number of files located in a single directory. As the number of the supported architectures and the number of assets in an Enterprise application increase, the number of object files and executable files in that directory will increase rapidly. Another disadvantage is due to the difficulties in maintaining the *Makefile* of an Enterprise application. Because of the increased number of object files located in the directory, the dependency relations may become excessively complicated and the *Makefile* eventually becomes hard to maintain.

5.2.2 Using Architecture-Specific Directories

An alternate way of distinguishing the different object files for the same asset is to group the files into different directories. This is the approach used in the current version of Enterprise. In this approach, sub-directories are created for every one of the supported architectures. The names of the sub-directories, such as "SUN4" and "MIPS", are used to identify the supported architectures, and the object files are put into their proper directories. For example, all the object files and executable files compiled for a SPARC workstation can be found in the "SUN4" directory. Similarly, the compiled code for a MIPS workstation can be found in the "MIPS" directory.

Since the object files compiled for different architectures are stored in separate directories, the naming problem of the object code and executable can be avoided. The naming of these files may now follow the same naming convention as used in sequential programming. This approach also has the advantage of not cluttering the original directory and makes the directory listing much easier to read.

Although the object codes of an Enterprise application are stored in different sub-directories, the source codes of the assets are still stored in the original directory of the Enterprise application. Symbolic links to the source modules are created in the sub-directories. On the one hand, we may perform the compilation in the source directory to avoid the need of the links, and move the object files to the proper directory when done. On the other hand, because the *Makefiles* used for building user programs may contain architecture-specific information (such as the pathnames of some libraries), the *Makefiles* used for each architecture may also differ slightly. The use of symbolic links to the source files has the advantage that it allows all architecture-specific files to be located in the sub-directories, and the architecture independent source files to be located in one source directory.

The use of symbolic links not only saves disk space by avoiding unnecessary duplication of the source modules, but also prevents any inconsistency among the source files if they are duplicated in different directories. This arrangement is particularly important from the debugging point of view. As it is comparably harder to debug a parallel program than

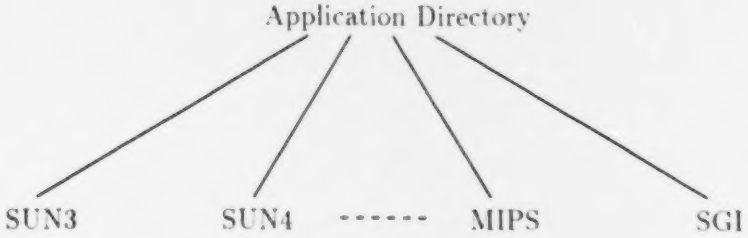

Figure 5.1: The Directory Structure of Enterprise

a sequential one, potential user errors, such as having an inconsistent set of source files, may make the debugging process even harder. It is often better to prevent the errors from happening rather than trying to eliminate the errors later [33]. The symbolic links to the source files prevent the possible error of having an inconsistent set of source files.

The overall directory structure of an Enterprise application is shown in Figure 5.1, where the "Application" directory stores the Enterprise graph and the assets' source code (".c" and ".h" files); the sub-directories store the object files and executable files of the Enterprise application for different architectures.

We have discussed, so far, how the source code and object code of an Enterprise application are organized. In the next section, we will describe the management of the makefiles of an Enterprise application.

5.3 Makefile Management for Enterprise Applications

As we discussed earlier, the object code of an Enterprise application is architecture dependent. The *Makefiles* used for managing the source and object files also may contain architecture-specific information. Because the executable copies of an Enterprise asset differ in both the compiled code and the libraries being linked, the *Makefiles* have to reflect these changes. The pathnames of the libraries and includes files, therefore, have to be changed slightly for each of the *Makefiles*. Similar to the object files, the *Makefiles* are put into the architecture-specific sub-directories.

Although we can specifically generate the makefiles for every sub-directory, it is much more convenient to use some readily available tools: *imake*, *makedepend*, and *xmkmf*. These tools not only help in generating makefiles in the sub-directories, but best of all, these tools provide exactly the service required in Enterprise.

Imake is a utility designed to eliminate the need to write makefiles directly [19]. It uses the C pre-processor, *cpp*, to generate the makefiles by using its include-file and macro-processing facilities. A directory-specific file, which is called the *Imakefile*, is used by the users to supply a specification to *Imake* describing how the programs in the directory are to be built. The specifications are usually written in the form of *Imake rules* (the rules will be described in a later section). The *Imakefile* is used together with some machine-dependent configuration files (Imake.tmpl, platform.cf, site.def, Project.tmpl and Imake.rules, which come from the X Window distribution) to create the appropriate *Makefile* for the programs in that directory.

Furthermore, *Imake* is meant to be architecture independent; it puts the architecture dependencies of the programs in one single location (the configuration files) to minimize the effort required in porting a program across different architectures. The *imakefiles*, used for generating conventional *Makefiles*, therefore, can be created in every one of the architecture-specific sub-directories of an Enterprise application. Each of these *imakefiles* differs slightly to reflect the different locations of the include files and the libraries on different workstations.

A *Makefile* is generated from an *Imakefile* by using *xmkmf*. *Xmkmf* is normally used to build X Window applications, but it can also be used to build other user-written programs from outside of the X Window source tree. When *xmkmf* is started from the command line, it tries to read in the *Imakefile* in the current working directory and calls *Imake* to generate the *Makefile* automatically.

To generate a proper *Makefile*, the dependencies of the object files and the header files have to be specified. Although the dependencies of the object files can be listed in the *Imakefile*, those of the header files cannot. The reason is that in the C programming language, the header files included by a source file are allowed to include other files. Besides, the system files to be included may also be organized differently on different systems. It will be hard for a user to list all of the header file dependencies manually. *Makedepend* is a tool which scans source files and include files to produce a list of header file dependencies suitable to be used in a *makefile*. By using *Imake*, a depend target may be created in the generated *makefile*. The command *"make depend"* can then be used to invoke *makedepend* to insert the header file dependencies into the *makefile* automatically.

In summary, a *Makefile* can be generated from an *Imakefile* by performing the following steps:

xmkmf
make Makefile
make depend

In the first step, *xmkmf* generates an initial *Makefile* which is used to generate the final *Makefile* for managing user programs. If a *Makefile* already exists in the directory, the original *Makefile* will be copied to *Makefile.bak* to avoid overwriting the original file. It is important to note that the command *xmkmf* must be executed on the target machine to ensure correctness of the *Makefile*, because the machine-specific information is read in when this command is being executed. The second step, *make Makefile*, generates the *Makefile* for managing or building user programs. Similar to the first step, if a *Makefile* exists, the file will be renamed as Makefile.bak. The last step, *make depend*, inserts the header file dependencies into the *Makefile* such that it will be able to build and manage user programs.

5.4 Creating the Sub-directories and Imakefiles

Although it is relatively easy to use the above steps to generate *Makefiles* for different architectures, it would be hard to create the appropriate *Imakefiles* manually. The implementation of the source and object code librarian includes a utility called *gen_make*, which takes a textual Enterprise asset graph as its input. *Gen_make* is designed for generating the required *Imakefiles* and for creating the necessary sub-directories for Enterprise applications.

The user of Enterprise can type in "*gen_make <application_graph>*" at the UNIX command prompt, and the program will create the sub-directories needed for different architectures, create one *Imakefile* in each sub-directory, and create in those sub-directories links which point to the source files. The user can then use *xmkmf*, and subsequent commands to create the *Makefiles* in the sub-directories.

The generated *Makefile* will be regarded as an input file to the commonly used *make* utility to manage user programs. The file contains a set of targets, and each target represents a sequence of actions to be executed for building or managing programs. When the command "*make <target>*" is issued, *make* will determine if the actions defined for the target have to be executed. For example, "*make Model*" will cause the asset Model to be compiled and built for execution.

After the *Makefile* is created, the normal *make* command can be used to build the executable copies of the assets in the Enterprise application. The request for compilation may come from either the execution manager or the user directly. The request can specify whether every asset in the directory is to be built or only some of the assets are to be built. The command "*make all*" could be used to specify that all assets in the directory should be built, if they do not exist or their source codes are modified. The command "*make <asset_name>*" is used to specify that the asset named as *asset_name* should be built. A clean target is also available in the *Makefile* if the user or the execution manager decides to clean up the directory. The clean up process generally involves the removal of all object files, executable files, and backup files in that directory.

5.4.1 Generation of the Imakefiles

The generation of the *Imakefiles* will be described in this section. The first step in generating an *Imakefile* is to read in the textual Enterprise graph. The textual graph contains the names of the assets to be built, the libraries to be linked in for different assets, the optional *CFLAGS* used in the compilation of an asset, and also other additional information. The *gen_make* utility parses the graph and saves the information for use in the subsequent steps. A more detailed description of the textual Enterprise graph can be found in Wong's thesis [52].

The *gen_make* program also reads in a configuration file that resides in the current directory. The file is named as *gen_make.cfg*. This file is used to specify the default values, for example the pathnames of the ISIS directories, used in the *Imakefile*. This allows the program to adapt itself easily to different environments. The format of the configuration file is similar to that of a conventional *makefile*. The default value is set by specifying the name of a variable, followed by an equal sign, and followed by the assigned value. For example,

ISIS_PATH = /usr/samson-pk/misc/distsys/src/isis/isisv2.1

specifies where the ISIS libraries can be found in the file system. The definitions of the variables can appear in any order in the configuration file. Furthermore, comments may be inserted into the configuration file by preceding them with the "#" character. Currently, the configuration file supports the definition of the following variables:

1. ISIS_PATH, specifies the path name of the ISIS libraries and include files which are needed for the compilations.

2. CC, specifies the default compiler to be used when compiling the assets.

3. CFLAGS, specifies the default *CFLAGS* options to be used. For example, the "*-O*" (optimize) option or the "*-g*" (debug) option.

An example configuration file is included in Appendix A.

The second step in generating the *Imakefiles* is to create the sub-directories for different architectures. The program first scans through the current directory to see if a directory or a file of the same name already exists. If such a name can be found, an error condition occurs; the named directory, as well as the *Imakefile* and symbolic links in that directory will not be created. The program skips that particular directory and proceeds to create the next one. If the directory can be created successfully, the program proceeds to the steps which create in the sub-directory an *Imakefile* and the symbolic links to the source files.

5.4.2 Contents of the Imakefile

A particular *Imakefile* created by *gen_make* can be divided into three sections. The first section contains the header information. This section is used to specify the values of the architecture-specific variables. Currently a variable *ARCH* is used to specify the architecture in that directory (e.g. $ARCH = SUN3$). This variable will be used in the subsequent sections of the *Imakefile* to further define other architecture specific values.

The second section of the *Imakefile* is the definition section. In this section, the pathnames of the libraries and include files are defined. Because the libraries are architecture specific, the pathnames of the libraries are defined by using *"$(ARCH)"* as part of the pathname definition. The value of *"$(ARCH)"* will be automatically expanded by *Imake* to specify the proper pathname to the libraries. Other definitions such as the system libraries (e.g. -lm for math libraries) and optional *CFLAGS* used in building the assets are also defined in this section. After these definitions, the names of the source files and object files for the assets are assigned to variables such as SRCS1 and OBJS1 (see Appendix B). All of these definitions will be used in the last section which contains a set of *Imake rules* to be described later. The variables defined in this section are listed as follows:

1. INCLUDES, this variable specifies the paths of the include files which may be scanned by *makedepend* to figure out the header file dependencies.

2. CC, this variable specifies the compiler to be used when building the assets. The default of the variable is to use the *cc* C compiler to compile the assets.

3. CFLAGS, this variable specifies the optional *CFLAGS* to be used when compiling the assets. The default value of this variable specifies the "-I" command line options of the *cc* compiler, such that the include files can be found correctly. The *CFLAGS* option specified in the application graph will be inserted before the default value.

4. ISISD, this variable specifies the pathname of the ISIS directory where the architecture specific ISIS libraries can be founded. An example entry will be (note: the pathname

is read from the configuration file, and *$(ARCH)* means that the value defined for the variable *ARCH* should be used):

ISISD = /usr/samson-pk/misc/distsys/src/isis/isisv2.1/$(ARCH)

5. ILIBS, this variable specifies the names of the ISIS libraries to be included. The value of the variable ISISD is used here to locate the libraries, such as "$(ISISD)/clib/libisis1.a".

6. SRCS, this variable specifies the names of the source files (i.e. the ".c" files). The variable will be used by *makedepend* to figure out the header file dependencies.

7. PROGRAMS, this variable contains the list of the names of every asset (executables) in an Enterprise application. The list is used in generating the *all* target, which allows all assets to be built by issuing *"make all"* at the command line.

The final section of the *Imakefile* contains *Imake rules* used to specify the actions to be generated in the final *Makefile*. An *Imake rule* is a predefined macro which will be expanded by the C pre-processor to produce the actions to be performed by a specific target. While some of these rules require variables to be passed to them explicitly, other rules operate on the values of the variables defined in the preceding section. Variables may be passed to the rules in a way that is similar to a C function call. If the number of variables being passed to a rule is less than that specified in the rule's definition, the sequence "/**/" may be used to replace a missing variable. Since the *Imakefile* will be passed through the C pre-processor, the sequence will be replaced by a NULL string and the missing variable will not be passed. Currently, three sets of rules are included in this section by *gen_make*.

The first set of rules is used to generate the actions for building the executable assets in the directories. The necessary actions are generated by using the *NormalProgramTarget()* rule. It is one of the predefined *Imake rules*, and is intended for building a single program (asset) in the current directory. One rule per asset is generated in the *Imakefile*, and the rules assume the following format:

NormalProgramTarget(program,objects,deplibs,locallibs,syslibs) where *program* is the name of the executable copy of an asset, *objects* contains the names of the object files needed to build the asset, *deplibs* are the libraries that have to exist before the linking process, *locallibs* are the libraries in the source tree to be linked in, and *syslibs* are the system libraries needed when building the asset. An example of the *locallibs* to be linked into an Enterprise asset would be the ISIS libraries.

The second set of rules contains a rule used to generate an *all* target in the final *Makefile*. This target allows every asset in a directory to be built by issuing the command *"make all"* at the UNIX command prompt. The *"AllTarget(programs)"* rule, another predefined *Imake rule*, is used to generate the necessary actions. The variable *programs* specifies the names of the assets to be built in the directory. To generate the actions correctly, the variable PROGRAMS defined in the earlier section is used, because the variable PROGRAMS is already assigned (with) the list of asset names in an earlier section. Therefore, the rule *"AllTarget($(PROGRAMS))"* is added into the *Imakefile*.

The final set of the rules is used to generate the depend target for invoking the *makedepend* command to find out the header file dependencies. A predefined *Imake rule*, *DependTarget()*, is used in this section to generate the required actions. This rule does not require variables to be passed to it explicitly. However, it assumes that a special variable, SRCS, has been set to contain the names of all the source modules (".c" files) used to build the individual assets. The value of this variable is assigned in the second section of the generated *Imakefile*, therefore, the rule *DependTarget()* rule is added to the *Imakefile* to create the depend target. A sample *Imakefile* created for a particular Enterprise graph (as described in Chapter 4) can be found in Appendix B.

5.5 Limitations of the Gen_make Utility

The current implementation of the *gen_make* utility does have a few limitations. First, the program is limited by the structure of the Enterprise application graph. The current structure does not provide a mechanism to specify the names of the local libraries and system libraries required by an asset separately. The libraries specified in the application graph by the library keyword are assumed to be system libraries.

There also does not exist a mechanism to specify some of the machine specific variables in the application graph. For example, when the animation example is compiled on a SUN 3/50 workstation, the assets have to be compiled with the *-f68881* option. This option can be specified in the application graph using the *CFLAGS* keyword and *gen_make* will handle it properly. On the other hand, if the animation example is compiled on a SPARC workstation, the *-f68881* now becomes an invalid option. This problem can be avoid by using symbolic *CFLAGS* instead of architecture-specific *CFLAGS*. For example, the "*-float*" flag may be used in the *CFLAGS* section in an Enterprise graph. The generation of the actual *CFLAGS* then depends on the target architecture. For a SUN 3/50 workstation, the *-float* may be translated into *-f68881;* while for a SPARC workstation, the option may be translated into an empty string. Other options such as the optimization flags and the debugging flags should also be defined symbolically. The expansion of symbolic *CFLAGS* has not been implemented in the current version of *gen_make*.

5.6 Conclusion

We have discussed in this chapter the problems encountered in managing the source code and object code of Enterprise applications. The naming problem of the object files and different copies of the same asset is solved by using sub-directories. The files are grouped into different directories according to the architecture for which the executable and object files are compiled. Although the executable and object files are located in different directories, the source code of the assets remains in the application's original directory. This is done to avoid the possible error of having an inconsistent set of source files.

The executable and object files in the sub-directories are managed by using normal *Makefiles*, and the generation of the *Makefiles* is handled by the *gen_make* utility. The utility reads in an application graph and generates an *Imakefile* in every sub-directory accordingly. *Imake* can then be used to generate the *Makefiles* needed to manage the executable and object files. On the one hand, the current implementation of *gen_make* does have some limitations, and it should not be difficult to remove these limitations in a future version. On the other hand, the *gen_make* utility also provides the users of Enterprise an easy to use utility to manage the source files and object files of Enterprise applications.

Chapter 6

Implementation of the Enterprise Compiler

6.1 Introduction

In an Enterprise application, we need to insert code into the user's source module to make a sequential program into a distributed program. The inserted code handles proper communication, synchronization, and other low-level communication protocols for the distributed program; this allows a distributed program to be built rapidly. In general, the insertion of the code can be accomplished either by using the pre-processor or the pre-compiler approach described in Chapter 2. The use of a pre-processor requires the users to insert in their code some additional keywords. These keywords are used to identify where and what code is to be inserted into the user's programs. The use of the latter approach can avoid the use of additional keywords at the expense of analyzing the context of the users' code and higher implementation cost. Both of these approaches have been used by other parallel programming tools. For example, FORCE [29, 27] and MONMACS [9] use the pre-processor approach, while Dino [40, 39], PAT [3], and MIMDizer [15, 50] use the pre-compiler approach [13].

The pre-processor approach is often implemented by using a low-level library, which implements a set of higher level parallel constructs. Compiler directives or macros are used to specify the location and the type of distributed code to be inserted into the users' source modules. With the help of the compiler directives, macros, and high-level parallel constructs, a parallel programming tool can then transform a sequential program into a distributed program. Although the distributed code is inserted into the source modules automatically by a pre-processor, users of the system still have to add in compiler directives or macros explicitly into their source modules. The languages used to develop the distributed program are often augmented by new keywords or macro definitions. Because of these new keywords, it is a common scenario that the users have to rewrite part of their sequential programs before they can be parallelized.

As noted by Chang and Smith [13], it is easier to implement the pre-processor approach than the pre-compiler approach. The latter approach is more difficult to implement because it often requires knowledge concerning the context of the user's source modules. Rather than using compiler directives or macros, a pre-compiler tries to analyze the source module and determines the locations of the distributed code automatically.

Both the inserted code and its locations are context dependent. For example, local variable A in function X is different from another variable A in function Y. The analysis process of a pre-compiler is often more difficult to implement. On the other hand, the pre-compiler approach requires less modification to the existing sequential programs before they can be parallelized by using a parallel programming tool. Moreover, users will be able to develop their new parallel programs by using a traditional and familiar sequential programming language, requiring less learning effort from the user's point of view.

6.2 Comparison between FrameWorks and Enterprise

While Enterprise is an evolution of FrameWorks, the compiler approach is used in Enterprise to replace the pre-processor approach used in the latter. The operations and the merits of both system will be discussed in the following sections.

6.2.1 The FrameWorks Approach

The general process of developing a FrameWorks application begins by separating a sequential application into different functional modules. Minor modifications to the each of these modules, such as adding the *call* statements as described in an earlier chapter, are needed and performed by the user to identify the distributed calls or replies between modules. A FrameWorks application graph is then created by the user to represent the parallel structure of his application. A pre-processor in FrameWorks, *fw_compile*, is then used to insert the appropriate distributed code into the user's source code to create a distributed application. Because of the pre-processor approach used by FrameWorks, two additional keywords are needed to augmented the syntax of the sequential programming language used. Users of the system have to use these additional keywords in developing their applications.

The keywords used in FrameWorks are *call* and *reply*. They are used to identify a remote procedure call and a return statement from a remote procedure respectively. Furthermore, users of the FrameWorks system have to package by hand the parameters of a *call* or a *reply* statement into *frames*. A *frame* is a structure in C which contains all the input/output parameters for a given *call* or *reply* statement. Although FrameWorks uses a pre-processor to take away the burden of handling proper synchronization and communication from the users, they are still responsible for the manual conversion of everyone of the distributed calls.

This approach has two major disadvantages. The first disadvantage is due to the added keywords to sequential programming language (C). FrameWorks has been successful in trying to reduce the number of modifications done to the programming language by using only two additional keywords. However, since the parallel structure of the program has been specified in the application graph, the keywords can be considered redundant. Furthermore, the *call* and *reply* statements imply a weak relation between the implementation of parallel procedures of a parallel program and its overall structure. The second disadvantage appears when the user wants to modify the parallel structure of the program. On one hand, it can be achieved by modifying only the application graph, if the design of the modules are left unchanged. On the other hand, both the application graphs and the *call/reply* statements have to be modified, if the user wants to change the design of the source modules (say changing from 3 modules to 4 modules). In addition, *call* and *reply* statements have an undesirable effect that if a parallel routine is now used sequentially, the statements have to be removed first.

One may think that it is possible to use a pre-processor, search for the name of the function call, and add in the *call* statement automatically. However, this is not true because the name of the function may appear anywhere in comments or in string constants. We do need a compiler in this case to recognize the location of function calls and return statements.

6.2.2 The Enterprise Approach

In the process of developing the Enterprise system, we have tried to improve on what had been achieved by FrameWorks. Our goal is to remove all keywords (*call* and *reply*) and save the user the hassle of making frames and making modifications to the program. By using the compiler approach, the *call* and *reply* statements can be removed. We can also eliminate the need for explicitly packaging parameters into frames. The symbol table in our compiler determines the parameter types in user written source code and packages the parameters automatically. In this way, users of Enterprise are now programming in traditional sequential C rather than dealing with augmented languages. The modules of an Enterprise application are normal C source files using the ".c" extension, not the ".f" extension as in FrameWorks, which can be compiled sequentially usually without any modification. One important advantage of this approach is the "real" separation of an application's distributed design and the source code used to implement the parallel procedures. The source code no longer contains any parallel structure information of the application. Instead, the information now resides totally in the Enterprise application graph. This separation allows an application to be changed easily in order to adapt to a dynamically changing environment such as a network of workstations.

6.3 Interface to the Compiler

The compiler used in Enterprise is actually a two-pass pre-compiler and is part of the Code Librarian. Both passes of the compiler are responsible for inserting the Enterprise code into user modules. In fact, almost all of the necessary Enterprise code is inserted during the first pass. The second pass is responsible for inserting Enterprise code to allow the use of *delayed blocking* or *future* variables described in an earlier chapter. The second pass is necessary only if there is a *f-call* in the module. To avoid using the value of a *delayed blocking* variable before it is returned from another module, *wait* statements must be inserted into the source code every time the value is accessed. The *wait* statement would block if the value has not been returned. Otherwise, the module may continue its execution and access the value safely.

The input to the Enterprise compiler is a source module and information on the parallel structure of the application (application graph). With this information, the compiler produces a new listing of the source module with the appropriate distributed code inserted. Then the new listing can either be passed on to the next pass, or in turn be compiled by a normal C compiler to produce an executable version of the program. The first pass of the compiler determines if there is a *f-call* in the module, and generates a shell script to handle the second pass of the compilation. Therefore, users do not have to decide if the second pass is necessary, but the system will perform the second pass automatically if it is needed (Figure 6.1).

6.3.1 First Pass of the Compiler

In the first pass, a "-F" option is used to specify the names of the distributed functions or function calls in that particular module. The compiler parses the whole module and inserts the necessary code at their proper locations, either transforming the named functions into distributed functions or distributed calls. In addition, if the distributed function returns a value to its caller, the first pass will also convert the *return* statement, packaging the return value into a message to be sent back. Because the code inserted into the module

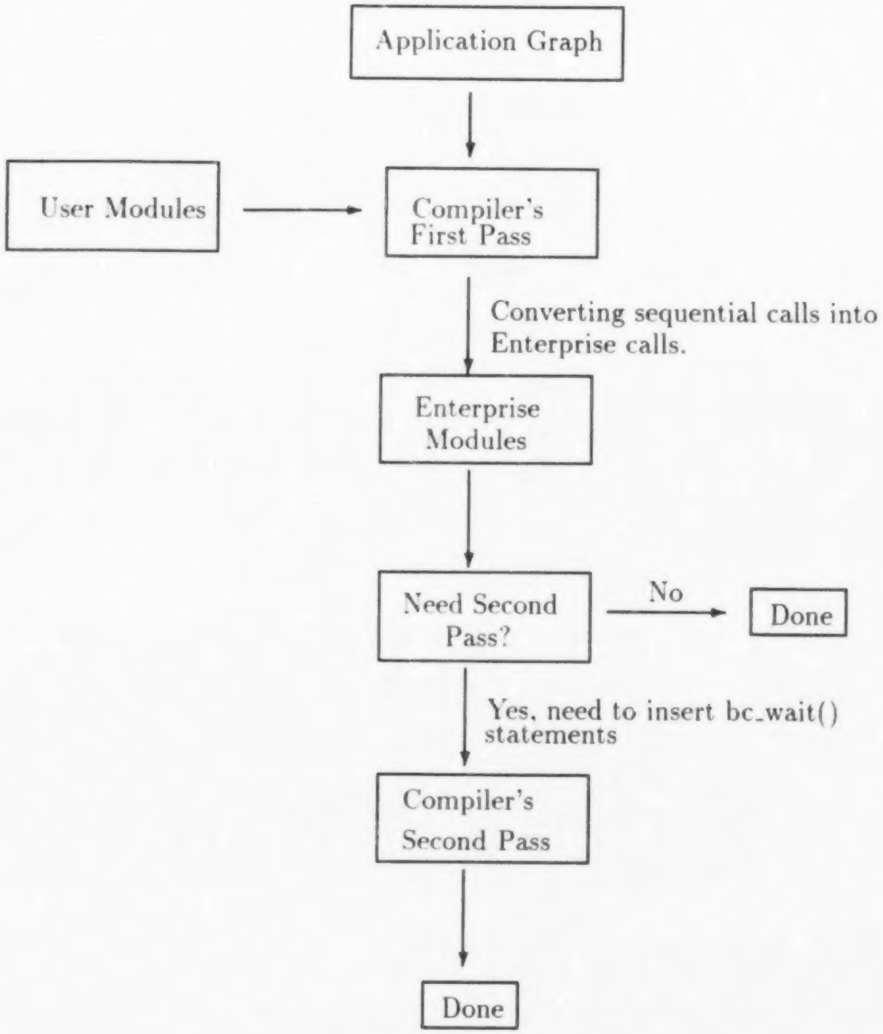

Figure 6.1: The Compilation of User Modules

also depends on the asset type of the function, the same option is also used to specify the asset type information. This is done by appending the asset types to the function names. Consider the animation example described in Chapter 4, if there is a call to "Split" in the module "PolyConv.c", and "Split" is configured as a pool. We can invoke the compiler by using:

```
Ent_gcc -FSplit.pool PolyConv.c
```

Currently, the Enterprise compiler supports the following asset types: Individual, Line, Department, Pool, Contract, and Service.

If there is more than one function to be converted in the module, a comma can be used to separate the different function names. Since in "PolyConv.c" the function "PolyConv" is called by other modules and is configured as a pool, we invoke the compiler by:

```
Ent_gcc -FSplit.pool,PolyConv.pool PolyConv.c
```

Note that the order of specifying the names of the interested function is not important. The output of the compiler will be put into a file called "_ent_input_filename" located in the users current working directory. For the example given above, a file called "_ent_PolyConv.c" will be generated.

An assumption is made by the compiler that if a function specified by the "-F" option is being called by other modules, then the definition (code) of the function also appears in that module. Otherwise, the option specifies the name of an Enterprise call made to other modules. Therefore, the need of the *call* and *reply* statements in FrameWorks can be safely eliminated.

Furthermore, the same pass is also used to identify the variables which are used for *f-calls*. *F-calls* is a new concept used in Enterprise. This concept allows all Enterprise calls to be translated into *non-blocking* distributed calls. For example, we could have the following statement in the module:

```
output = F (input);
```

While FrameWorks will translate the statement into a *blocking* call to function *F*, Enterprise translates the statement into a *non-blocking* Enterprise call (*f-call*) of *F*. The calling module will continue to execute until the value *output* is needed but the called module has not yet returned. The *delayed blocking* is achieved by adding a *wait* statement every time before the variable is accessed. The statement blocks, if the output value have not yet returned; otherwise, the module will proceed to the next statement and continue to execute. For example, consider the following code segment, which makes an Enterprise call to function *F*, puts the return value into variable *output*, and the return value is not used right after the call:

```
output = F (input);
/* some other code */
result = output + 1;
```

will be translated into:

```
output_token = call F (input);
/* some other code */
wait (output_token);
result = output + 1;
```

Note that the code listed in the above example is only the pseudo-code of what is actually generated by Enterprise. The actual code is much more complex, and will be described in a later section of this chapter. The *output_token* in the example above is an integer allocated by Enterprise. For every variable used for receiving a value from a *f-call*, an integer must be declared to hold the status of the variable. Its status tells Enterprise if the return value has been returned.

If there is a loop in the program, the value of *output* may actually be accessed after, but appeared before the call statement. The insertion of the *wait* statements, therefore, is performed in the second pass to avoid semantic ambiguity. Although the statements are inserted in the second pass, the first pass must record the names of the variables used in a *f-call* (for *delayed blocking*) and the names of the functions where the variables are declared. Both of these names are needed to distinguish between other variables with the same name but declared in other functions belong to the same module. This information is passed on to the second pass of the compiler where the insertion of the *wait* statement is performed.

The last function of the first pass of the compiler is to output a *check file* named as the same module name but ended with the ".chk" extension. For example, the check file of "test.c" will be called "test.chk' in the same directory. This file is used for checking the consistency of Enterprise calls and replies. Since if a module makes a *f-call* to another module and the latter module does not reply, then a deadlock situation may occur. Conversely, if a function makes replies to a *p-call*, the replies will be queued up and may cause a program to exit abnormally. To avoid these undesirable effects, a utility called *Ent_check* is used to parse the generated check files and issues warnings if it finds any inconsistent Enterprise calls. Because an Enterprise program is separated into different modules, the consistency checking can only be performed after each of the modules has been compiled. The utility reads in the application graph to identify the names of the check files and to perform the consistency check. Two kinds of information are included in the check files:

1. Declaration of an Enterprise function or procedure: If the subroutine is a procedure, i.e. it does not return a value (but a statement like "return;" is allowed in a procedure), it is classified as a *P_PROC*. The output of the compiler will be an entry in the file started with the keyword *P_PROC*, followed by the name of the procedure, and then the number of parameters specified in the declaration. For example if PolyConv is a procedure there are two parameters specified in the declaration, then the entry will be:

   e.g. `P_PROG PolyConv 2`

   If the subroutine is a function, the keyword will be replaced by *F_PROG*, and the rest of the information remains the same.

   e.g. `F_PROG PolyConv 2`

2. Asset calls information: Depends on the type of the call, it can be either a *P_CALL* or a *F_CALL* in the check file. The format of the entry will be the keyword *P_CALL*

or *F_CALL*, followed by the name of the caller, then the name of the called function, and finally the number of parameters specified in the call. For example, if procedure X makes a P_CALL to Y and specifies two parameters in the call, then the entry becomes:

```
P_CALL X Y 2
```

Similarly, if the call is a f_call, then the entry will be

```
F_CALL X Y 2
```

An example of a generated ".chk" file can be found in Appendix C.

6.3.2 Second Pass of the Compiler

The second pass of the compiler is invoked by the "-V" option. The option is used to specify the names of the variables and the functions in which they are declared. The syntax of the option is similar to the "-F" option used in the first pass. Again if there is more than one variable, a comma can be used to separate each pair of the variables and function names. Consider the case when function "PolyConv" returns a value back to "Model" and the variable *result* appears in the function "Model", that is we have in module Model:

```
result = PolyConv();
```

then we will invoke the second pass by:

```
Ent_gcc -VModel.result _ent_Model.c
```

Note that because we would like to leave the user-written source code intact, the input to the second pass is the temporary listing file generated by the first pass of the compiler. The output file of the second pass is named as "_ent2_Model.c" and can be compiled by using a normal C compiler. As mentioned earlier, a shell script is generated to handle the second phase of compilation automatically. After the second pass is performed, the shell script also renames the "_ent2_Model.c" file back to "_ent_Model.c" to avoid any confusion. The *delayed* *blocking* used in Enterprise helps to improve the resource utilization and the efficiency of a distributed application. However, because of semantic issues which will be discussed later, some restrictions have to be imposed on the properties of the variables that can be appeared on the left-hand-side of a *f-call*. These restrictions will be discussed in the semantic issues section.

6.4 Implementation Details

The compiler used in Enterprise is actually a modified version of the GNU gcc (version 1.38) compiler [48]. Two options are added to the compiler for inserting code into Enterprise modules. A "-F" option is used to invoke the first pass of the Enterprise compiler, and a "-V" option is used to invoke the second pass of the compiler.

6.4.1 The GNU C Compiler

To implement a C compiler from scratch is fairly expensive. We chose to implement our precompiler on top of the GNU C compiler because it is in the public domain and the possibility of reduced implementation cost. By using the GNU C compiler, we can also take advantage of its symbol table to extract the necessary type information. Most of the modifications done on the compiler are in the files "toplev.c" and "c-parse.y". The file "toplev.c" contains the code used to accept the new options. The file "c-parse.y" contains the C grammar rules, actions, and procedures used to insert the code needed for distributed processing.

6.4.2 Implementation Procedures

On our first attempt, we tried to use only the grammar of the GNU C compiler by stripping out all the actions in the grammar file (c-parse.y). Once the actions were removed, we discovered that the GNU C compiler uses the symbol table built by the compiler to distinguish between an "identifier" and a "type" declaration. Therefore, a lot of the actions in the grammar file are needed for the compiler to parse a C program correctly.

To preserve the correctness of the compiler, we decided to leave the actions in the file intact and add in new actions for inserting Enterprise code. This results in a slower compilation because each modules must be compiled two or three times to produce the corresponding executable files. On the other hand, this approach allows us to construct a working compiler in a shorter time span. The new actions in the grammar file have to perform the following functions:

1. Recognizes the name of the function that is being parsed and the name of the function that is being called.
2. Recognizes the return statements in a function.
3. Recognizes the types of the parameters and constants used in a function call.
4. Recognizes the name of the variable which appears on the left-hand-side of an assignment statement followed by an Enterprise call. The compiler has to distinguish between a *f-call* and an Enterprise call without a return value (i.e. a *p-call*).
5. Recognizes array references and, if possible, calculates the size of a given array.
6. Inserts ISIS [7] code accordingly. As mentioned in Section 4.6, ISIS is used as the back-end of Enterprise.

The implementation of the above functions will be discussed and described in details in the following subsections.

6.4.3 Recognition of Function Names and Function Calls

The first thing that the Enterprise compiler needs to perform is to build a table of the function names to be recognized. The table is built before the parsing of a module is started. The asset types of the functions are stored in the same table. In most situations, few functions will need to be recognized in a particular module. Therefore, a dynamically created link list is used to store the table of function names, and the list is searched by linear probing. The link list allows unlimited number of function names to be stored in the table. The definition of the link list is:

```
struct flist {
    char    *name;        /* function name */
    char    *type;        /* asset type    */
    struct  flist  *next;
};
```

There are two variables in the GNU C compiler which allow us to access the name of the function or function call that is being parsed. The variable, in GNU C, *current_func_decl* contains the name of the function that is currently being parsed. The variable *lastiddecl* contains the last identifier encountered which, depending on the context, can be the name of a function call. Whenever a function definition or a function call is encountered, its name is checked against the table of function names. If the names match, Enterprise code is inserted into the new listing automatically.

An example of the user-written source code and the code inserted for the Animation example is included in Appendix D and Appendix E respectively.

6.4.4 Recognition of *RETURN* Statements

Whenever a *return* statement is encountered in a given module, the Enterprise compiler checks if the *current_func_decl* exists in the table of function names. If the name exists, then the compiler is parsing an Enterprise function which sends replies back to its caller. The type of the return variable is checked and an Enterprise statement used to return the value is added accordingly. Only the type of the return variable is used in constructing the reply messages. The Enterprise compiler does not check the consistency between the type of the return variable and the declared type of the function. However, since the original actions of the GNU C compiler are left intact, any inconsistency will trigger the GNU C compiler to issue a warning. The generated listing can still be compiled successfully, but the correctness of the program cannot be guaranteed.

6.4.5 Recognition of Parameter Types

The type of the parameters, of an Enterprise call or a function declaration, are extracted from the symbol table built by the GNU C compiler. Because of the representation used by the GNU C compiler, some type information cannot be easily extracted. Different procedures are needed to extract the type of, say a character array, user defined structure, simple parameters, and the type of an expression. This type information is needed to package the parameters into frames or messages to pass to other modules. Two approaches were developed to do the packaging of parameters automatically.

The first approach conforms to the ISIS standard. In the ISIS documentation, a parameter in an Enterprise call has to have a corresponding type character to specify its type. This is similar to the *printf()* statement commonly used by C programmers. For example if an Enterprise call is made to another module and *cntr* is an integer, the call

```
F (cntr);
```

Will be translated into:

```
call F ("%d", cntr);
```

For simplicity reason, we use *call F* to represent the translated Enterprise call. In the actual generated code, the call will be translated into either a call to *bcast()* or *bcast_l()*. The two functions are ISIS library calls, where the first one uses synchronous broadcast and the second one uses asynchronous broadcast, to send a message across the network. The "%d" is used in ISIS to specify that "cntr" is in fact an integer. ISIS uses a lower case letter to specify a single element, and an upper case of the same letter to represent an array of the element. The number of elements in the array is needed in the latter case. There are two special cases in the use of the formatting characters in ISIS. The formatting character used for a single element of either the type *float* or *double* must be treated as a single element array. The definitions of the type characters is listed in Table 6.1 (where size is equal to number of elements in the array):

Variable Type	*Single Element*	*Arrays*
int	%d	%D[size]
long	%d	%D[size]
short	%h	%H[size]
float	%F[1]	%F[size]
double	%G[1]	%G[size]
char	%c	%C[size]
others	User defined	User defined

Table 6.1: Definitions of type characters in ISIS.

As in the table, if there is a user defined type or a structure in the module, and a variable of such a type is used as a parameter in an Enterprise call, ISIS requires a new letter be assigned for the type. The same letter should be used whenever the user defined type is used again in a call or a reply statement. User defined type can be created by using *typedef* statements in C. New types are frequently used for one of two purposes. First, the new types can be used for documentation purpose and provide "mnemonic synonyms for existing predefined, derived and user-defined types"[30]. The new types can increase the readability of a program by grouping related variables into a single structure. For example, a structure of a student record may contains information such as name, age, identification number, and other related information. Second, new type names may be used to hide hardware details. For example, on some machines an integer of type *int* may be a 32 bits integer, but on other machines the type *long* may be needed to represent the same integer. By using *typedef* statements, the changes can be localized in only one statement to handle the changes in hardware [30].

To make sure that the letter for a given type is common for each module, a file ".ent.def" is used to record the user defined types. When the Enterprise compiler comes across a new user defined type, the name of the structure is added to that file. The file is read in before the compilation starts, such that the letters used in earlier compilation can be used to describe the same types again.

On the other hand, the existence of the ".ent.def" file creates some possible concurrency problems. If two modules are being compiled concurrently, which is perfectly acceptable if the modules do not depend on each other, then the ".ent.def" may be accessed and updated by the two compilation processes at the same time. Even if only one process has the right to write to the file at any single time instance, both processes may have chosen the same letter to represent two different user defined types. Therefore, the only solution is for the Enterprise compiler to read the whole ".ent.def" file every time before it attempts to update the file. This solution is expensive and inefficient. Moreover, there are only a limited number of letters which can be used to describe user defined types in a program using ISIS. Only 14 letters can be used for the new types, because other letters are predefined in ISIS and cannot be reused in this manner.

6.4.6 Another Approach to Handling Parameter Types

Because of such disadvantages as a limited number of user defined types and the concurrency problems described in the previous section, another approach is used to handle the parameter types. The second approach, although not mentioned in the ISIS manual, solves our earlier problems. The normal base types in C are still handled by the earlier method as outlined above, but user defined types are treated differently.

User defined types are treated as arrays of characters instead. The length of the character array is the total number of bytes of the specified type, if the variable to be passed is a single element. This length can be calculated easily by using the *sizeof()* function in C. If the variable is an array of a user defined type, then

Length of Array = Number of elements * sizeof (user_defined_type).

The number of elements is obtained by using the symbol table generated by the GNU C compiler. By using this method, more than 14 user defined types are allowed in an Enterprise program and the concurrency problem is eliminated as well. This method is used in the current version of the Enterprise compiler.

As of the current version of Enterprise, the issue of byte ordering and the way that different compiler allocates memory for the variables have not yet been addressed. For example, a nine-byte structure on a SUN SPARC workstation requires twelve bytes to store the structure, but on a SUN 3/50 workstation the structure requires only ten bytes of memory to store it. The difference in storage spaces is due to the memory alignment schemes used by different hardware. Furthermore, different compiler may organize the members in a structure differently. Currently, the ISIS package used by Enterprise has no direct support for exchanging data between heterogeneous computers. One possible solution to the problem is by using Sun's XDR (eXternal Data Representation) library routines to package the variables. The XDR routines can be used to package arbitrary data structures into a hardware-independent manner. The possibilities of using these routines together with the current back-end have not been explored due to resource constraints.

6.4.7 Recognition of Variables for Delay Blocking

In a statement, such as

```
output = F (input);
```

the call to function F is processed by the compiler before the "=" operator is detected. In other words, we need to store the name of the function call somewhere, so that our compiler can determine whether or not the variable *output* is being used in a *f-call*. The name of the function call *F* is stored in a variable called *func_name* when the compiler is running. The translation of a function call into an Enterprise call can assume one of two forms. The actual translation depends on whether or not the call returns a value to its caller. Thus, the translation is performed after the compiler determines if an "=" sign exists, but not immediately after the function call is recognized.

6.4.8 Interface to ISIS Code

This section is used to describe the actual code inserted by the compiler to transform a sequential call into an Enterprise call. In the previous sections, we described the translation of a function call by adding the *call* keyword in front of the corresponding function call. However, this is different from the actual generated code.

Enterprise Calls

Since ISIS is used as the back-end of Enterprise, the code generated for the *f-call* and *p-call* are either ISIS library calls, or calls to functions that are built on top of ISIS. Enterprise calls are translated into different forms of the ISIS library calls: *bcast()*, or *bcast_l()*. An Enterprise call which does not return a value, that is a *p-call*, is translated into:

```
bcast (_e_name, _e_CALL, format, parm, _e_NREPLY);
```

Where *_e_name* is the string "_e_" followed by the function name, *format* is the formatting string for the parameters *parm*, and *_e_NREPLY* specifies that the call is indeed a *p-call*. For example the function call *F(x)*, where *x* is an integer and no reply is expected, is translated into:

```
bcast (_e_F, _e_CALL, "%d", x, _e_NREPLY);
```

An Enterprise call which returns a value, that is a *f-call*, is translated into a call to *bcast_l()*. The function *bcast_l()* is used because it allows the idea of *delayed blocking* to be realized. When compared to *bcast()*, the function *bcast_l()* has a higher communication cost, but the function also increases the potential parallelism of the program.

As mentioned in Section 6.3.1, the realization of *f-call* requires an integer token to be declared for every outstanding return values. The tokens can either be declared statically at compile time, or allocated dynamically at run time. If the tokens are declared statically, a lot of memory storage is required. Consider the example "MAT = F(x)", and MAT is an integer matrix of 512 × 512. To ensure correct access to the matrix, every element of the matrix must be assigned with a unique token. In other words, an additional of 262,144 integers need to be added to the program. However, only a small portion of the tokens may actually be used by the program. These tokens may quickly become a major factor in the amount of memory for doing real computations.

The approach used in Enterprise is by using a dynamically created data structure to store the addresses of *delayed blocking* variables. Since efficiency is not yet a major concern in the current implementation, a linked list is used to store the addresses. However, other data structures, such as a tree or a hash table, can be used later to improve the efficiency when it becomes a factor affecting the overall performance of an application. An element in the linked list contains the address of a *delayed blocking* variable, the token for telling the return status of the variable, and a link to the next element in the list. An Enterprise supplied function *_ent__insert()* is used to create and add new elements into the linked list. New elements are always added at the head of the list.

An Enterprise call which returns a value, that is a *f-call,* is translated into:

```
_ent__addr = & out_parm;
_ent__insert (_ent__addr);
_ent__list->token = bcast_l ("f", _e_name, _e_CALL, format1, in_parms,
                             _e_REPLY, format2, _ent__addr);
```

where *_ent__addr* stores the address of the return variable *out_parm,* the function *_ent__insert()* inserts the address of the variable into the linked list, *format1* is the formatting string for the input parameters *in_parms, _e_REPLY* specifies that the call is a *f-call, format2* is the formatting string for the output parameter *out_parm,* and finally *_ent__list→token* is the token used to store the return status of the variable.

The use of the variable *_ent__addr* is used to prevent possible side-effects. It is possible in C that *out_parm* is an expression by itself, such as the statement "x++ = F(x,y)". If we simply attach the *address of* operator (&) to the expression on the left-hand-side of the equal sign, use the new expression in place of *_ent__addr* in the second and third statements, then x will be incremented twice producing erroneous results.

For example, the *f-call* "$x{+}{+} = F(x,y)$", where x and y are integers, is translated into:

```
_ent__addr = & x++;
_ent__insert (_ent__addr);
_ent__list->token = bcast_l ("f", _e_F, _e_CALL, "%d%d", x, y, _e_REPLY,
                             "%d", _ent__addr);
```

In an earlier version of the Enterprise compiler, only the ISIS function "bcast_l" is used to construct both the *p-calls* and the *f-calls*. The different types of calls are differentiated by using the "_e_REPLY" and "_e_NREPLY" keywords. However, this approach was replaced by the current version to reduce the unnecessary overhead involved in making *p-calls*.

Enterprise Wait Statements

The Enterprise wait statements must be inserted into a function that makes *f-calls* to other Enterprise modules for the function to execute correctly. The statements are inserted every time before a ***delayed blocking*** variable is being accessed. The insertion of the statements is similar to the translation of a *f-call*, where the address of the variable is first stored in the Enterprise variable *_ent__addr*. The variable is again used to avoid undesirable side-effects. The code inserted for accessing integer x, say "y = x + 1", will be:

```
_ent__addr = & x;
_ent__ptr = _ent__lookup (_ent__addr);
y = *(int *) _ent__addr + 1;
```

In the generated code, the function *_ent__lookup()* tries to find an entry containing the same address of x. If such an entry is found, the function will block until the variable is returned; when it returns, the entry will be deleted from the link list. If a corresponding entry cannot be found in the link list, then the value is already returned and the program may proceed safely. In the last generated statement, the variable x is replaced by the expression **(int *) _ent__addr* for the same reason as described in the previous section. The type used in the expression (int) and the type of the original variable must agree with each other. Therefore, the compiler is responsible for finding the type of the variable and generating the correct expression accordingly.

Enterprise Return Statements

Return statements, used in a function which may be called by a *f-call*, to return a value back to its caller is translated into:

```
reply_l ("f", msg_p, format, parm, _e_NREPLY);
return;
```

For example the statement "return x;", where x is an integer, is translated into:

```
reply_l ("f", msg_p, "%d", x, _e_NREPLY);
return;
```

The additional *return* statement is needed to force the function being called to exit properly. Otherwise, the function may continue to execute after sending the message and may create unexpected results.

In some cases, a function that is called by a *p-call* may contain return statements in it. These return statements do not reply a message back to its caller, but simply cause the function to exit. To handle this scenario, instead of using the *reply_l ()* statement, a simple *return* statement is inserted into the user's source code. Note that the *reply* statement is designed to return a value to its caller without knowing the name of the caller. The details concerning the implementation of each asset and the use of ISIS library calls can be found in Wong's thesis [52].

Function declarations

The function declaration of an Enterprise entry procedure is also modified by the Enterprise compiler. Although the function is written in sequential code, the function declaration must be changed such that it can receive parameters sent by remote procedures. To allow a function to use received messages as input parameters, the function header must be changed to accept only one input parameter of the ISIS defined type *message*. The required parameters are then extracted from the message by using the *msg_get()* function in ISIS. This function will be inserted as the first statement to be executed by the entry procedure. The function has a similar format to the *sscanf()* function in C, except that the values are extracted from a message using the ISIS defined formatting characters.

As an example, the following function:

```
test (choice1, choice2)
int  choice1;
int  choice2;
{
  /*
   * Local variable declarations.
   */

  /*
   *  Code to be executed by the program.
   */
}
```

will be translated into:

```
/*
 * test (choice1, choice2)
 */
test (msg_p)
message  *msg_p;
{
int  choice1;
int  choice2;
{
  /*
   * Local variable declarations.
   */

  msg_get (msg_p,"%d%d", &choice1, &choice2);

  /*
   *  Code to be executed by the program.
   */
}
}
```

Notice that the original declaration of the function is commented out by the compiler and extra brackets are inserted to maintain the correct syntax in C. Appendixes D and E contain the source code of the Animation example before and after it is compiled by the Enterprise compiler to show the differences.

6.5 Semantic Issues

Although the programming language (C) used in Enterprise has no additional keywords, it has been augmented by new semantics. The obvious changes in semantics can be found in the *f-call* and *p-call* described above. The new semantics allow users to develop distributed applications in a way that is similar to writing sequential programs. The low-level communication protocols will be handled by the system automatically. These semantics, however, require some restrictions to be imposed on the use of some variables. The restrictions are needed to ensure the correctness of a parallel program.

One of the restrictions is that a variable, called as a "call_var", which appears on the left-hand-side of a *f-call* cannot be a global variable. Consider the following example:

```
A()                        main()
{                          {
  x = 3;                     x = F(1);
}                            A();
                             printf ("%d", x);
                           }
```

If *F()* is an Enterprise call, then depending on the timing, the *printf* statement may actually print out the value 3 or the value of *F(1)*. The non-deterministic result is because *wait* statements will be inserted into the *main* program only. When the second pass of the compiler is invoked, the option "-Vmain.x" will be used for the above example. The option tells the compiler to insert *wait* statements for variable x into the *main* program. However, when the variable is reassigned in procedure A, the procedure does not check if the values is returned or not. If the value was returned before the execution of function A, then x will be assigned the value of 3; otherwise, the value of x will be overwritten by the value of *F(1)*. Since it is not easy for the compiler to deduce from the above example the intention of the programmer, it is not possible to ensure the correct execution of the program. A sophisticated control flow analysis is needed to deduce the required information, but most compilers do not provide such a mechanism. To avoid the ambiguity, a global variable cannot be used as a "call_var".

The second restriction is that address aliasing cannot be performed on a "call_var". This restriction is designed to avoid the problem of non-deterministic result due to address aliasing. In Enterprise, only the name of a variable rather than the address of a variable is recorded. A C programmer, however, can use address aliasing to reference the same variable by using variables of different names. The effect of address aliasing can also be changed from time to time during the execution of a program and causes the program to generate non-deterministic results.

Furthermore, if the address of a "call_var" is used to keep track of the status of the variable, problem of non-deterministic results may still occur. Consider the following example:

```
A()                  B()                  main()
{                    {                    {
  int x;               int y,x;             A();
                                            B();
  x = F (3);           y = G (1);         }
}                      x = y + 1;
                     }
```

If both *A()* and *B()* are local functions, and both *F()* and *G()* are *f-calls*, then, depending on the timing, y may get the value of *F(3)* but not *G(1)*. The reason for this effect is due to the way that a C program manages its memory. In a normal C program, when function A is being called, the memory needed to accommodate the function's variables is pushed onto a stack. The memory is popped from the stack when the function exits, and the memory needed for the next function is then pushed onto the same location. In the above example, when function A is finished and function B starts its execution, the memory used for the variable y in function B has the same address of the variable x used in function A. As a result, the value of y cannot be determined until run time. To avoid this problem, all outstanding *f-calls* are cancelled upon the return of local procedures. The calls are cancelled by emptying the list which stores the addresses of ***delayed blocking*** variables. The effects of *f-calls,* therefore, are limited to within the scope of a single procedure.

There are still other semantic issues yet to be resolved. One example is the semantics of loops and arrays. In the following code segment:

```
for (i = 0; i < 10; i++)
    a[i] = F(i);

/* other user code */

for (i = 0; i < 10; i++)
    b[i] = a[i];
```

if *F(i)* is a *f-call*, then the second for-loop in the example will continue to execute until all of the calls to function *F* are returned. Even though there are no semantic ambiguities, the performance of the program may be decreased. As it stands, the for loop will wait for the a[i]'s sequentially, that is a[0], a[1], ..., a[9] to return from *F(i)* regardless of their timings; but during execution, a[9] may actually be returned first with a[0] returned last. This shows the undesirable effect of having to wait for the results to return in sequence. This situation should be handled by an ***asynchronous pool*** asset which uses a statement like "array_wait(a, i)". The statement waits for the first value returned from a *f-call*, and updates the value of "i" and "a[i]" accordingly.

6.6 Conclusion

The overall cost of compiling an Enterprise module is relatively high. Part of the cost is due to the fact that the Enterprise compiler is built on top of the GNU *gcc* compiler. Therefore, a given module may in fact be compiled three times in order to produce a corresponding executable. While this increases the compilation time, this approach provides us a quick way to produce a working prototype of the compiler. Also, because of the general availability of the GNU *gcc* on many different hardware platforms, it should be easy to port the compiler onto other computers (the current version was developed and tested on Sun 3 and Sun 4 workstations running Sun OS 4.1.1).

There are some other limitations on the use of parameter types for an Enterprise call resulting from the use of ISIS as the back-end. First, ISIS limits the number of user defined types allowed in an Enterprise program. Second, ISIS has no direct support for passing multi-dimension arrays. These limitations have been removed in Enterprise by casting user defined types and multi-dimension arrays into single dimension arrays. However, the problem of byte ordering and memory alignment problems have not be solved in the current implementation. Finally, the code generated by Enterprise, and the semantic ambiguities resulted from the use of *delayed blocking* variables were described in this chapter.

Chapter 7

Conclusions and Future Research

7.1 Introduction

This thesis presented the design and the development of part of a new programming environment, called Enterprise, for writing distributed applications. The Enterprise code librarian is implemented as part of this thesis. The code librarian consists of a source and object code management utility, and a compiler used to parallelize user programs. The source and object code management utility manages the object files of Enterprise applications while taking into account the possibility of heterogeneous network of workstations. The Enterprise compiler relieves the user from programming the low-level communication protocols. The user writes a distributed program using the sequential C language, and specifies the parallelism of the program through a structured asset graph. The compiler then performs the parallelization, according to the asset graph, by inserting additional distributed code into the user's program.

Although Enterprise is still in its prototyping stage, major components of the system have been implemented. They are the graphical user-interface [12, 49], the execution manager [52], and the code librarian. These components allow programs to be constructed using the Enterprise model and provide the means to assess the usefulness of the system. Several applications have been developed using Enterprise, and experimental results have been gathered. The results showed that the system requires programmers to think and program in a less conventional manner, but the new manner is relatively easy to learn and master.

7.2 Thesis Summary

This thesis includes the description of the model, the design of the architecture, and the implementation of tools to support the system. The main emphasis of this thesis is on the development of some of these tools. The first part of the thesis describes the Enterprise programming model and the architecture of the system. The system is divided into well-defined subcomponents such that the functions of each subcomponent can be specified and be implemented separately. The remaining part of the thesis presents the design and implementation of the code librarian of the system.

The earlier part of the thesis describes the Enterprise programming model, the design principles of the system, and the architecture of the system. Enterprise is designed as a high-level programming model for writing distributed programs. It has the unique feature of allowing the total separation of application code and parallel structuring information. The application code itself is written in familiar sequential languages, and the structuring information is specified in a hierarchically organized asset graph. The Enterprise graph is at a much higher level when compared to those being used in other parallel programming tools. Users no longer draw the graph by connecting processes with communication channels. Instead, the graph is constructed by using two basic operations: coercion and expansion. These operations are used in Enterprise to support the notion of parallelization by partitions and replications.

An organization analogy is used in Enterprise to describe the structures of distributed applications. This analogy has the benefit of allowing a consistent set of names to be used to describe the commonly used parallel constructs. These constructs are called *assets* in the analogy, and they represent high-level parallelization techniques such as *lines* or *departments.* Chapter 4 explains this analogy and describes the meaning of the supported assets.

The chapter also presents the architecture of Enterprise and outlines the functions of each subcomponent. Several subcomponents have been implemented to allow programs to be constructed and to gather initial experimental results.

The later half of the thesis presents the design and implementation of the Enterprise code librarian. The librarian includes a utility for managing the source and object code of Enterprise applications. It is implemented using the *Imake* [19] utility. The librarian manages different binary executable files by grouping them into architecture-specific sub-directories. An *Imakefile* is created in each sub-directory to generate the *Makefile* uscd for managing the executable and object files. Chapter 5 describes the implementation of the source and object code librarian in detail.

A compiler is also developed as part of the librarian for inserting low-level communication codes into user programs. The codes that are being inserted are calls to the ISIS distributed library package [7]. The Enterprise compiler recognizes and generates codes for remote procedures, remote procedure calls, and return statements from remote procedures. It is also capable of packaging the parameters of a module call into messages to be passed across a network. The current compiler is implemented by modifying the GNU gcc compiler.

In Enterprise, no additional keywords are needed in the user-written sequential code. The use of a compiler showed that a distributed program can be separated into two independent parts: the sequential source code, and the parallel structure. This total separation of application code and parallel structuring information allows the user to restructure a finished application without modifying the source code. Furthermore, existing sequential programs can be parallelized easily because of this separation. The removal of additional keywords provides an easy way for automatic sequential to parallel transformation and vice versa. Finally, the user is also encouraged to experiment with different configurations for an application. Our initial experience with Enterprise showed that distributed applications can be built quickly using the environment and reasonably good speedup can be achieved. Chapter 6 of this thesis presents the implementation of the Enterprise compiler and discussion of related semantic issues.

7.3 Recommendations for Future Research

Although the major components of Enterprise have been implemented, the whole system is far from complete. Research is still taking place to refine the system. For example, the definitions of the assets are being revised, the graphical interface is being enhanced, test cases are being developed, and possible extensions are being considered. This section presents some of the possible refinements and enhancements to the system.

1. Currently, only several components of the Enterprise system have been implemented. For example, the graphical user interface is partially functional and still under construction. The current resource secretary chooses the workstations by checking only their load averages. A more capable resource secretary should provide more detailed selection criterion, such as the processor speed or the amount of installed memory, for choosing the suitable workstations. Currently, research effort is actively taking place in both implementing and refining the Enterprise system.

2. In general, we need to re-compile an asset if its source code has been modified, or if the role of an asset has been coerced. Furthermore, not every coercion operation, such as coercing from a *pool* to a *contract*, requires re-compilation of the asset. Changing a *pool* to a *contract* changes only the runtime environment of the program, but not the source code of the program. The code librarian handles the re-compilation of an asset if its source code has been changed. However, it is not easy to determine if an asset has been coerced in the current textual Enterprise graph. Stronger support for determining when to re-compile an asset is needed.

3. The makefile generation utility is limited by the use of machine-specific compilation flags. It needs a mechanism for specifying the flags, such as *-f68881*, symbolically. The utility can be extended to accept the symbolic flags and to expand them according to the intended architecture of the program.

4. One of the goals in designing Enterprise is to provide transparent access to heterogeneous computers on a network. Since different hardware uses different schemes of byte-ordering and memory alignment schemes, Enterprise applications currently run only on a homogeneous network of workstations. The ISIS library package handles the byte-ordering problem but not the memory alignment problem. The compiler can be improved to handle the different schemes automatically. One possible solution is by using SUN XDR (eXternal Data Representation) to handle the memory alignment problem.

5. The Enterprise model limits the types of variables that may be passed from one process to another. The model only supports the passing of non-pointer type variables. However, supporting pointer type variables should prove to be a useful feature because they are used frequently in C programs. The current compiler can be enhanced to allow the passing of pointers by copying the contents of the pointer to a message explicitly. The major difficulty in handling pointers is determining the size of the contents. A C programmer can easily allocate a large block of memory by using *malloc()*. A compiler, however, cannot correctly determined at compile time the size of the memory being allocated. To generate the correct code for passing pointers, the user must specify the pointer to be passed and the total number of bytes to be passed in an Enterprise call explicitly. One way to support the passing of pointers is to impose the restriction that a pointer argument in an Enterprise call must be followed by an argument specifying the size of the contents.

6. Enterprise can be extended to accommodate other programming languages. The implementation of the current compiler is based on the GNU C compiler which also contains grammar rules for the language C++. In other words, it may be possible to modify the Enterprise compiler to handle user's C++ programs without too much trouble. However, a shorter compilation time could be expected if the Enterprise compiler is redeveloped from scratch. The improved performance could provide a stronger support for compilation-on-demand.

7. As discussed in Chapter 4, several new asset kinds are under construction. The semantics of a *asynchronous pool*, a *static division*, and a *dynamic division* is still under active research. Additional asset kinds may also be introduced to allow greater flexibility in structuring applications. Since each asset kind is a self-contained entity, adding new asset types to Enterprise can be done in an easy way.

8. The low-level communication protocols are currently implemented using the commercial version of ISIS libraries. Although a public domain version of ISIS is available, its many limitations forced us to use the commercial one. This implies that users must install ISIS before they can use Enterprise to write programs. In fact, the current version of ISIS still imposes some limitations on the applications. Using other communication libraries may be a potential solution to the limitations, but the fact that there still does not exist a *standard* package for handling the low-level protocols makes choosing the right package a difficult task.

Bibliography

[1] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, 1986.

[2] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transaction on Programming Languages and Systems*, 10(1):51–86, January 1988.

[3] Bill Appelbe, Kevin Smith, and Charlie McDowell. Start/pat: A parallel-programming toolkit. *IEEE Software*, pages 29–38, July 1989.

[4] O. Babaŏglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex: An environment for parallel programming in distributed systems. Technical Report UB-LCS-91-01, University of Bologna, 1991.

[5] Robert G. Babb II, Lise Storc, and Robert Hiromoto. Developing a parallel Monte Carlo transport algorithm using large-grain data flow. *Parallel Computing*, 7:187–198, 1988.

[6] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn Mckinley, and Jaspal Subhlok. The parascope editor: An interactive parallel programming tool. *ACM*, pages 540–550, November 1989.

[7] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual Version 2.1.* Computer Science Department, Cornell University, 1991.

[8] K. S. Booth, J. Schaeffer, and W. M. Gentleman. Anthropomorphic programming. Technical Report CS-82-47, Department of Computer Science, University of Waterloo, 1982.

[9] James Boyle, Ralph Butler, Terrence Disz, Branett Glickfeld, Ewing Lusk, Ross Overbeek, James Petterson, and Rick Stevens. *Portable Program for Parallel Processors.* Holt, Rinehart and Winston, Inc.

[10] J. C. Browne, Muhammad Azam, and Stephen Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, pages 10–18, July 1989.

[11] James C. Browne, Taejae Lee, and John Werth. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Transactions on Software Engineering*, 16(2):111–120, February 1990.

[12] Enoch Chan, Paul Lu, Jimmy Mohsin, Jonathan Schaeffer, Carol Smith, Duane Szafron, and Pok Sze Wong. Enterprise: An interactive graphical programming environment for distributed software development. Technical Report TR 91-17, Department of Computing Science, University of Alberta, June 1991.

[13] Long-chyr Chang and Brian T. Smith. Classification and evaluation of parallel programming tools. Technical Report CS90-22, University of New Mexico, 1990.

[14] A. Chatterjee. A mechanism for concurrency among objects. In *Proceedings of Supercomputing '89*, pages 562–567, 1989.

[15] Richard H. Chill. MIMDizer – A new tool for parallelization. *Supercomputing Review*, pages 26–28, April 1990.

[16] CYBER 205 supercomputer introduction. R515.1187, Computing Services, University of Alberta, 1987.

[17] E. E. Dijkstra. *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, New York, 1982.

[18] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[19] Paul DuBois. Using Imake to configure the X Window System Version 11, Release 4. Wisconsin Regional Primate Research Center, August 1990.

[20] S. I. Feldman. Make – A program for maintaining computer programs. *Software Practice and Experience*, 9:255–265, 1979.

[21] Ian Foster and Stephen Taylor. Strand: A practical parallel programming tool. Technical Report MCS-P80-0889, Argonne National Laboratory, 1989.

[22] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1989.

[23] Jason Gait. A distributed process manager for an engineering network computer. *Journal of Parallel and Distributed Computing*, 4:423–437, 1987.

[24] Warren Harrison. Tools for multiple-cpu environments. *IEEE Software*, pages 45–51, May 1990.

[25] R. Jagannathan, A. R. Downing, W. T. Zaumen, and R. K. S. Lee. Dataflow-based methodology for coarse-grain multiprocessing on a network of workstations. In *1989 International Conference on Parallel Processing*, 1989.

[26] A. Jones and P. Schwartz. Experience using multiprocessor system - a status report. *Computing Surveys*, 12(3):121–166, June 1980.

[27] Harry F. Jordan. The Force. Department of Electrical and Computer Engineering, University of Colorado at Boulder.

[28] Harry F. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 86.

[29] Harry F. Jordan, Muhammad S. Benten, Norbert S. Arenstorf, and Aruna V. Ramanan. *Force User's Manual.* Department of Electrical and Computer Engineering, University of Colorado at Boulder.

[30] Stanley B. Lippman. *C++ Primer.* Addison-Wesley Publishing Company, 1991.

[31] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring distributed applications as fragmented objects. Technical Report Rapport de Recherche INRIA 1404, INRIA, January 1991.

[32] T. A. Marsland, T. Breitkreutz, and S. Sutphen. NMP – A network multi-processor. Technical Report TR-88-22, Department of Computing Science, The University of Alberta, December 1988.

[33] C. E. Mcdowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 2(4):593–622, 1989.

[34] Myrias. *Parallel Programmer's Guide.* Myrias, Research Corporation, Edmonton, Alberta, Canada, 1990.

[35] David A. Nichols. Using idle workstations in a shared computing environment. *Comm. ACM*, pages 5–12, November 1987.

[36] Cherri M. Pancake. Conceptual models in parallel programming languages. In *Proceedings ACM Southeast Region 26th Annual Conference*, pages 369–375, 1989.

[37] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. *ACM*, pages 627–636, November 1989.

[38] Terrence W. Pratt. The PISCES 2 parallel programming environment. In *Proceedings of the International Conference on Parallel Processing*, pages 439–445, 1987.

[39] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. The Dino parallel programming language. Technical Report CU–CS–457–90, Department of Computer Science, University of Colorado at Boudler, April 1990.

[40] Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver. Massive parallelism and process contraction in Dino. Technical Report CU–CS–467–90, Department of Computer Science, University of Colorado at Boudler, March 1990.

[41] Zary Segall and Larry Rudolph. PIE: A programming and instrumentation environment for parallel processing. *IEEE Software*, pages 22–37, November 1985.

[42] J. F. Shoch and J. A. Hupp. The Worm programs-early experience with a distributed computation. *Comm ACM*, 25(3):172–180, March 1982.

[43] Ajit Singh. *A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations*. Ph. D. Thesis, Department of Computing Science, University of Alberta, March 1991.

[44] Ajit Singh, Jonathan Schaeffer, and Mark Green. Structuring distributed algorithms in a workstation environment: The FrameWorks approach. In *Proceedings of the International Conference on Parallel Processing*, pages 89–97, August 1989.

[45] Ajit Singh, Jonathan Schaeffer, and Mark Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[46] G. Singh and Mark Green. Visual programming of graphical user interfaces. In *Workshop on Visual Languages*, pages 161–173, Linkoping, Sweden, August 1987.

[47] G. Singh and Mark Green. A high-level user interface management system. In *ACM SIGCHI'89*, New York, April 1989.

[48] R. M. Stallman. *Using and Porting GNU CC (for V. 1.35)*. Free Software Foundation, Inc., 1989.

[49] D. Szafron, J. Schaeffer, P. S. Wong, E. Chan, P. Lu, and C. Smith. The Enterprise distributed programming model. In *Working Conference on Programming Environments for Parallel Computing*, International Federation for Information Processing, Edinburgh Parallel Computing Centre, April 1992.

[50] Joel Williamson. MIMDizer tools aid Fortran code redesign. *IEEE Software*, page 50, May 1990.

[51] Pok Sze Wong. A study of the conceptual computation models of parallel programming environments. CMPUT 507 Project, Department of Computing Science, University of Alberta, 1991.

[52] Pok Sze Wong. The Enterprise Executive. M. Sc. Thesis, Department of Computing Science, University of Alberta, 1992.

Appendix A

The Configuration File of Gen_make

An example of the gen_make configuration file, gen_make.cfg, read by gen_make.

```
#####################################################
#
#  Configuration file used by gen_make.
#
#####################################################

ISIS_PATH    =  /usr/samson-pk/misc/distsys/src/isis/isisv2.1
CC           =  cc
CFLAGS       =  -I/usr/therien/operations/src/X11R4/contrib/toolkits/xview2/
build/usr/include \
                -I/usr/samson-pk/misc/distsys/src/isis/isisv2.1/include \
                -I../../ -g
```

Appendix B

The Generated Imakefile

The Imakefile generated by gen_make and the textual graph described in Chapter 4:

```
####################################################
#
#  Imakefile created for an Enterprise application.
#
####################################################

ARCH    =    SUN3
INCLUDES=    -I/usr/samson-pk/misc/distsys/src/isis/isisv2.1/include \
             -I../../
CC      =    cc
CFLAGS  =    -f68881 \
-I/usr/therien/operations/src/X11R4/contrib/toolkits/xview2/build/usr/include \
-I/usr/samson-pk/misc/distsys/src/isis/isisv2.1/include \
-I../../ -g
ISISD   =    /usr/samson-pk/misc/distsys/src/isis/isisv2.1/\$(ARCH)
ILIBS   =    \$(ISISD)/clib/libisis1.a \$(ISISD)/clib/libisis2.a \
             \$(ISISD)/mlib/libisism.a

SRCS1   =    _ent_Model.c
OBJS1   =    _ent_Model.o
LIBS1   =    -lUTILITY -lfb -lm

SRCS2   =    _ent_PolyConv.c
OBJS2   =    _ent_PolyConv.o
LIBS2   =    -lUTILITY -lfb -lm

SRCS3   =    _ent_Split.c
```

```
OBJS3   =    _ent_Split.o
LIBS3   =    -lUTILITY -lfb -lm

SRCS    =    _ent_Model.c _ent_PolyConv.c _ent_Split.c
OBJS    =    _ent_Model.o _ent_PolyConv.o _ent_Split.o
PROGRAMS=    Model PolyConv Split

# Rules to compile an Enterprise application.

  NormalProgramTarget (Model, $(OBJS1), /**/, $(ILIBS), $(LIBS1))

  NormalProgramTarget (PolyConv, $(OBJS2), /**/, $(ILIBS), $(LIBS2))

  NormalProgramTarget (Split, $(OBJS3), /**/, $(ILIBS), $(LIBS3))

# Define all the targets for the programs

  AllTarget($(PROGRAMS))

# Generate the dependencies rules for makedepend.

  DependTarget()
```

Appendix C

The Generated Check File

This appendix includes an example check file that is generated by the consistency check utility described in Chapter 6. This check file is generated for a test program, and it contains a number of inconsistent calls. If this file is passed to the utility, errors will be reported. The generated check file is listed as follows:

```
F_PROC test 1
P_PROC set_name 4
F_CALL main test 1
F_CALL main set_name 0
F_CALL main test 1
P_CALL main test 1
P_CALL main set_name 2
```

Appendix D

Example of User Source Code

This appendix shows the pseudo code of the Animation example. For the sake of readability, only the main procedure calls of *Model*, *PolyConv*, and *Split* are shown.

Asset Code: *Model*

```
/* Model asset */

#include "fish.h"

#define NUMBER_STEPS 4
#define NUMBER_FISH  10
#define NUMBER_FRAMES 20

Model()
{
  float  timeperframe;
  int    frame;
  int    result;

  /* Generate the school of fish */
  MakeFish (NUMBER_FISH, 0);

  /* Loop through each frame */
  timeperframe = 1.0 / NUMBER_STEPS;
  for (frame = 0; frame < NUMBER_FRAMES; frame++)
  {
    /* Do model computations */
    InitModel (NUMBER_FISH);
    MoveFish (NUMBER_FISH, timeperframe);
    DrawFish (NUMBER_FISH, timeperframe * frame);
    WriteModel (frame);

    /* Done!  Send work to PolyConv process */
    result = PolyConv (frame);

    if (result == -1)
      exit(-1);
  }
}

main()
{
  Model();
}
```

Asset Code: *PolyConv*

```
/* PolyConv asset */

#include "fish.h"

#define MAX_POLYGONS 1000

PolyConv (frame)
int       frame;
{
  polygon polygontable [MAX_POLYGONS];
  int       npoly;

  /* Convert polygons and send to Split */
  DoConversion (frame);
  npoly = ComputePolygons (polygontable);
  Split (frame, npoly, polygontable);
}
```

Asset Code: *Split*

```
/* Split asset */

#include "fish.h"

#define MAX_POLYGONS 1000

Split (frame, npoly, polygontable )
int       frame, npoly;
polygon polygontable;
{
  HiddenSurface (frame, npoly, polygontable);
  AntiAlias (frame, npoly, polygontable);
}
```

Appendix E

Example of Enterprise Inserted Code

This appendix shows the code inserted into the user's source code as described in Chapter 6. This section contains only the code that is inserted by the Enterprise compiler, but not the header code generated by the Enterprise executive [52]. The code of the animation example after it is processed by the Enterprise compiler is listed as follows (Enterprise inserted code are listed in *italic*):

Asset Code: *Model*

```
/*
 * Additional declarations added in by
 * Enterprise for handling futures.
 */
#include                 ''Ent_list.h''

_ent_List                _ent_list = NULL;
_ent_List                _ent_ptr  = NULL;
unsigned int             _ent_addr = 0, _ent_dummy = 0;


/* Model asset */

#include "fish.h"

#define     NUMBER_STEPS      4
#define     NUMBER_FISH       10
#define     NUMBER_FRAMES     20

Model()
{
   float     timeperframe;
   int       frame;
   int       result;

   /* Generate the school of fish */
   MakeFish (NUMBER_FISH, 0);

   /* Loop through each frame */
   timeperframe = 1.0 / NUMBER_STEPS;
   for (frame = 0; frame < NUMBER_FRAMES; frame++)
   {
      /* Do model computations */
      InitModel (NUMBER_FISH);
```

```
    MoveFish (NUMBER_FISH, timeperframe);
    DrawFish (NUMBER_FISH, timeperframe * frame);
    WriteModel (frame);

    /* Done!  Send work to PolyConv process */
    /*
     *  Function call of PolyConv()
     *         result = PolyConv (frame);
     */
    _ent_addr = &        result
    _ent_insert (_ent_addr);
    _ent_list->token = bcast_l(''f'',_e_PolyConv,_e_CALL,''%d'',frame,_e_REPLY,''%d'', _ent_addr);


    /*
     * Waiting for reply
     *    if (result == -1)
     */
    _ent_addr = &result;
    _ent_ptr = _ent_lookup (_ent_addr);

    if (*(int *) _ent_addr == -1)
      exit(-1);
  }
}

main()
{

  /*** Enterprise code inserted for main() ***/
  _e_init ();

  Model();

  /* ISIS code inserted for exit in "main". */
  exit(0);
}
```

Asset Code: *PolyConv*

```
/*
 * Additional declarations added in by
 * Enterprise for handling futures.
 */
#include               ''Ent_list.h''

_ent_List              _ent_list = NULL;
_ent_List              _ent_ptr  = NULL;
unsigned int           _ent_addr = 0, _ent_dummy = 0;


/* PolyConv asset */

#include "fish.h"

#define     MAX_POLYGONS       1000

/*
 * This is an Enterprise asset.  The original
 * declaration of the function is commented out
 * and is replaced by the inserted code.
 *
 * PolyConv (frame)
 */
PolyConv(msg_p)
message *msg_p;
{
int            frame;
{
   polygon polygontable [MAX_POLYGONS];
   int         npoly;

   /* Convert polygons and send to Split */

   /*** ISIS code inserted.                                        ***/
   /*** Asset type : Pool                                          ***/

   msg_get(msg_p,''%d'',&frame);

   DoConversion (frame);
   npoly = ComputePolygons (polygontable);
   /*
    *   Function call of Split()
    *       Split (frame, npoly, polygontable);
    */
   bcast(_e_Split,_e_CALL,''%d%d%C'',frame,npoly,polygontable,sizeof(polygontable *)*1000,_e_NREPLY);


}     /*** Enterprise inserted bracket.                                ***/
}
```

Asset Code: *Split*

```
/*
 * Additional declarations added in by
 * Enterprise for handling futures.
 */
#include               ''Ent_list.h''

_ent_List              _ent_list = NULL;
_ent_List              _ent_ptr  = NULL;
unsigned int           _ent_addr = 0, _ent_dummy = 0;


/* Split asset */

#include "fish.h"

#define     MAX_POLYGONS       1000

/*
 * This is an Enterprise asset.  The original
 * declaration of the function is commented out
 * and is replaced by the inserted code.
 *
 * Split (frame, npoly, polygontable )
 */
Split(msg_p)
message *msg_p;
{
int         frame, npoly;
polygon     polygontable;
{

   /*** ISIS code inserted.                                            ***/
   /*** Asset type : Pool                                              ***/

   msg_get(msg_p,''%d%d%C'',&frame,&npoly,&polygontable.&_ent_dummy);

   HiddenSurface (frame, npoly, polygontable);
   AntiAlias (frame, npoly, polygontable);

}    /*** Enterprise inserted bracket.                                  ***/
}
```